Computer Science

TECHNICAL REPORT

The Use of Data Flow Information for the Selection and Evaluation of Software Test Data

P. Frankl

Technical Report 394

August 1988

NEW YORK UNIVERSITY

[redacted]

The Use of Data Flow Information for the Selection and Evaluation of Software Test Data

P. Frankl

Technical Report 394

August 1988

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

This work was supported in part by the National Science Foundation under Grant NSF DCR 8[illegible]

To the memory of my aunt, Paula F. Silver

The Use of Data Flow Information for the Selection and Evaluation of Software Test Data

Phyllis G. Frankl

Elaine J. Weyuker, Research Advisor

Two families of software test data adequacy criteria, each based on data flow analysis, are defined for programs written in Pascal. Their formal properties are investigated and interactive software testing tools based on them are described.

The first of these families, the data flow testing criteria, was previously defined for programs written in a simple language. We extend the definitions to apply to programs written in Pascal. The data flow testing criteria are based purely on the syntax of the program being tested. They require that the test data execute certain paths from program points at which variables are defined to program points at which those definitions are used. We describe the design and implementation of a software testing tool, ASSET, based on the data flow testing criteria

A serious weakness of the data flow testing criteria is that for some programs there exists no set of test data which is adequate for testing the program according to these criteria. This problem arises due to unexecutable paths in the program The second family of criteria, the feasible data flow testing criteria, circumvent this problem by eliminating from consideration those definition use associations which can never be exercised We show that certain formal properties of the feasible data flow testing criteria differ significantly from those of the data flow testing criteri

Since it is undecidable whether a given set of test data satisfies a given feasible data flow testing criterion, feasible data flow testing cannot be fully automated. However, it can be partially automated. We describe a heuristic method, the path expression method, which attempts to determine whether a given definition-use association can be exercised. The path expression method is based on a combination of data flow analysis and symbolic evaluation. We introduce a new symbolic evaluation technique which is more general, but essentially no more expensive, than symbolic execution. The path expression method, along with ASSET, constitute a tool which partially automates feasible data flow testing.

Acknowledgements

I would like to thank my advisor, Elaine Weyuker, for her invaluable technical guidance and moral support. Stewart Weiss, Deberah Rennels, Ernie Campbell, Valerie Barr and Eugene Rodolphe participated in many useful discussions of this work. I would also like to thank the many friends who put up with me while I worked on this thesis. Finally, I would like to thank my parents, Daniel and Estelle Frankl, who provided unwavering support and encouragement throughout my education.

Table of Contents

APPENDICES

CHAPTER I:

Introduction

1. Background

The development of techniques for testing programs is among software engineering's most perplexing problems. Ideally, these techniques should be amenable to automation and should yield test sets which are small enough to be useful in practice yet which are likely to expose errors in the program being tested. Unfortunately, there are many theoretical and practical obstacles to the development of such techniques.

It is often stated that the goal of testing is the *detection of errors*. That is, the goal of testing is to find an input which causes the program's output to differ from the specified output. This distinguishes testing from *program verification* in which one attempts to mathematically prove the correctness of the program. Of course, if the program has no errors, then the testing process can never achieve this goal. A more realistic, though less precise summary of the aim of testing is, "the goal of testing is to detect errors *or, failing that, to increase confidence that the program is correct*."

To test a program P, one chooses a set T of possible inputs to P, runs P using the elements of T as inputs, and compares actual outputs to the expected outputs. If an error is found, then the testing phase terminates and the debugging phase begins. If no error is found, one can either choose additional test data and continue the testing phase, or can release the software to the user. If T is deemed to be a "good" test set, then the fact that running the program on the elements of T has failed to expose errors in the program should increase confidence that the program is correct.

Two key questions in software testing research are, how should software test data be selected?" and "how can one tell whether a program has been tested 'enough'?". A *test data selection method* is a procedure for choosing test cases for a program. A *test data adequacy criterion* is a predicate used to *evaluate* the test data, that is, to determine whether a program has been tested "enough". Given a test data selection method M and an adequacy criterion A, we can summarize the testing process with the following algorithm:

```
begin
    initialize test set T to the empty set;
    error_detected := false;
    adequately_tested := false;
    while (not error_detected) and (not adequately_tested) do
    begin
        if T is adequate for testing program P whose specification is S
            according to adequacy criterion A
        then adequately_tested := true
        else
        begin
            choose set $T'$ of additional test cases
                according to test data selection method M;
            run P on all inputs from $T'$, comparing outputs
                to the specified outputs;
            if error is detected then error_detected := true;
            T := $T \cup T'$
        end
    end
    if error_detected then report error
    else release program P, certifying that it has been adequately tested
        according to criterion A;
end;
```

It is possible to define test data selection methods and test data adequacy criteria in completely *ad hoc* ways. It seems reasonable, however, to expect that test data selection methods and adequacy criteria which exploit available information will yield test sets which are more likely to expose those errors which exist in P.

There are several sources of information available to aid in the selection and evaluation of software test data. These include the program text, the program's specification, and information about common types of errors. Those techniques which are based primarily on the program text are called *program based*, those based on the specification are called *specification based*, and those based on information about common errors are called *error based*. Because program specifications are often somewhat vague documents written in natural language, and because of the scarcity of information on "common errors", program based methods are currently more amenable to automation than are specification based or error based methods. The software testing strategies examined in this thesis are program based.

Our emphasis in this thesis will be on adequacy criteria. Note however that there is a correspondence between test data selection methods and test data adequacy criteria. Corresponding to an adequacy criterion C, there is a test data selection method M_C, which, in effect, says, "select a test set which satisfies criterion C". For example, if the adequacy criterion says that the test data must cause the execution of every statement in the program, then the corresponding selection method would involve examining the program to find inputs which cause the execution of particular statements. Similarly, corresponding to a selection method M, there is an adequacy criterion method C_M, which says, "A test set is adequate if it has been produced by method M". Thus, while this thesis will mainly discuss adequacy criteria, the reader should bear in mind that these criteria also implicitly correspond to test data selection methods.

2. Terminology

Let U be the set of numbers and characters representable on some particular computer system. The *input space*, I, and *output space*, O, are both equal to the set of finite sequences of elements of U. A specification is a relation $S \subseteq I \times O$. The set $D^S = \{i \in I \mid \exists\, o \in O$ such that $(i,o) \in S\}$ is called the *input domain of specification S*. A program P defines a partial function (which by abuse of notation we will also call P) $P : I \rightarrow O$. The *input domain of program P* is set $D^P = \{i \in I \mid$ program P halts on input $i\} = \{i \in I \mid P(i)$ is defined$\}$. Those inputs which cause program P to crash or to go into an infinite loop are not in D^P.

Program P is correct with respect to specification S if and only if

1) $D^S \subseteq D^P$, and

2) $\forall i \in D^S((i,P(i)) \in S$.

That is, P is correct with respect to S if and only if for each element i of the input domain of S, P halts on input i, returning a value which is in accordance with the specification.

A test set T for a program/specification pair, (P,S) is a subset of the specification's input domain. Throughout this thesis, when we address the question of whether a test set T satisfies an adequacy criterion, we will also assume that T is a subset of the program's input domain. This is a reasonable assumption because one normally applies an adequacy criterion after the program has run successfully on all elements of the test set. When no ambiguity results, we will omit reference to the specification S and assume that $D^P \subseteq D^S$.

A program unit's structure can be represented abstractly by its *flow graph*, a single-entry, single exit directed graph in which each node represents a sequence of statements which are always executed as a unit and each edge represents a conditional or unconditional transfer of control. A *path* is a sequence $(n_1, \ldots, n_m)$ of nodes such that (n_i, n_{i+1}) is an edge for $1 \leq i < m$. A *complete path* is a path $(n_1, \ldots, n_m)$ in which n_1 is the entry node and n_m is the exit node of the flow graph. If G is the flow graph of a program unit P, we will sometimes refer to a path through G, as a *path through P*. In chapter 2 we will give a more precise definition of the flow graphs for the class of program units we are considering.

3. Limitations of Testing

We now take note of some of the inherent limitations of program testing as a validation strategy.

We first note that it is usually impractical to prove a program correct by testing it. Let T be a proper subset of the input domain, D^S of a specification S. Let $t_0 \in D^S - T$ be a point such that for some y, $(t_0,y) \notin S$. Then there is a program P such that $(t,P(t)) \in S$ for all $t \in T$ but $(t_0,P(t_0)) \notin S$. P is incorrect with respect to S, but this fact is not exposed by testing P on the test set T. Thus the only way to prove a program correct by testing it is by testing it on every element of the D^S (except perhaps those elements i for which $\{y|S(i,y)\}=U$).

Another limitation of testing stems from the fact that in order for testing a program to be meaningful, there must be some way of determining for each test case t whether $(t,P(t)) \in S$. This may be a difficult problem, for example, if the amount of data produced by the program is very large, or if the program is written to "find the answer" to some question for which the answer is not known [WEY82].

Undecidability results place limitations on the automatability of program based testing. A node, branch, or path is *executable* if and only if there is some element of the input space which causes it to be executed. The following undecidability results place fundamental limitations on the feasibility of automating the program validation process

[WEY79,DAV83]: There is no algorithm to determine whether

(1) a given program P halts on a given input t,

(2) a particular statement, branch, or path is executable,

(3) a particular path from the entry to the exit of a program is executable.

Problem (1) is the famous halting problem; the halting problem can easily be reduced to the problems in (2); the unsolvability of (3) follows from the unsolvability of Hilbert's Tenth Problem [DAV73].

4. Related Work

Several software test data adequacy criteria are based on the idea that one cannot consider a program to be adequately tested if certain sequences of statements have never been executed by any test data. These methods, known as *structured testing*, associate a *test set* T, with the set Π of paths through program P's flow graph which are executed when P is run with inputs from T. The test set T, or equivalently the set of paths Π, is C-adequate for P if and only if each of the sequences required by C is a subpath of one of the paths in Π.

The strongest of the structured testing criteria is path testing, which requires that every path through the program's flow graph be executed by the test data. Path testing is impractical for all but the most trivial programs, since any program having a loop has infinitely many paths. Nevertheless, it is useful to study path testing because some of the limitations of path testing apply *a fortiori* to the other structured testing techniques.

One limitation of path testing is that it cannot guarantee the detection of so-called *missing path errors* [GOO75]. If the program fails treat some special case, there will be no path in the program which corresponds to that special case, and hence the test data will not be required to exercise that special case. In fact, a similar problem plagues all program based testing strategies: if the program lacks some feature which it is supposed to have, analysis of the program text will fail to suggest the inclusion of test data aimed at exercising that feature.

A second limitation of path testing arises due to *coincidental correctness*. Suppose that test cases t and t' both cause path π to to be executed, and that $(t,P(t)) \in S$, while $(t',P(t')) \notin S$. Then a test set T which includes t but not t' will execute path π without exposing the error. For example, suppose that path π is supposed to read a number, square it, and print the result; but instead it reads a number, doubles it, and prints the result. Then the input t=2 will execute π but fail to expose the error.

Statement testing and branch testing are well-known structured testing methods which are weaker than path testing. Statement testing requires that every statement (equivalently, every node in the program's flow graph) be executed, while branch testing requires that every edge in the program's flow graph be executed. These criteria are based on the intuition that one cannot feel very confident that the program is correct if some statement or branch has never been executed by any of the test data. Unfortunately, statement and branch testing can fail to expose many common errors [HOW78,HUA75]. Several criteria which are based on analysis of the program's control flow and which are stronger than branch testing but weaker than path testing have been proposed [HOW78,MIL74,WOO80].

More recently, a number of test data adequacy criteria which are based on data flow analysis, some of which "bridge the gap" between branch testing and path testing, have been proposed and studied [HER76,RAP82,RAP85,LAS83,NTA84,CLA85]. Tools based on some of them have been implemented [FRA85a,FRA85b,GIR85,KOR85]

These criteria are based on the intuition that one should not feel confident that a variable has been assigned the correct value at some point in the program if no test data causes the execution of a path from the assignment to a point where the variable's value is subsequently used.

Rapps and Weyuker [RAP82,RAP85] defined a family of criteria based on data flow analysis for programs written in a simple language having only simple variables, input/output statements, assignment statements, and conditional and unconditional branching. The basic idea of these criteria is to require the test data to exercise certain paths from points in the program at which a variable is defined to points at which that definition is used. Rapps and Weyuker compared the various criteria in this family to one another and to statement, branch and path testing. The family of criteria defined in chapter two of this thesis is obtained by extending the definitions of the Rapps and Weyuker criteria to apply to programs written in a large subset of Pascal.

The criterion defined by Herman is similar to one of the Rapps and Weyuker criteria, the "all-uses" criterion. Roughly speaking, The criteria defined by Ntafos [NTA84] requires the test data to exercise sequences of length k of definition-use pairs, where a variable x_1 is defined in node i_1 and used in node i_2, and variable x_2 is defined in node i_2 and used in node i_3, etc. The criteria defined by Laski and Korel [LAS83] are based on the intuition that the value assigned to a variable at a particular node n may depend on the values of several variables, each of which could possibly have several definitions which reach node n. The criteria require the test data to exercise paths along which various combinations of these definitions occur. Clarke et. al. [CLA86] compared the criteria defined by Rapps and Weyuker, by Ntafos and by Laski and Korel to one another.

An adequacy criterion C satisfies the *applicability property* if and only if for every program P there exists some test set which is C-adequate for P [WEY86]. One would expect a "good" adequacy criterion C to satisfy the applicability property. However, none of the structured testing criteria satisfy the applicability property. The problem arises due to the fact that some path which is required by the criterion may be unexecutable. Furthermore, for each of the structured testing criteria, it is undecidable whether a test set exists which adequately tests a given program. In chapter 4 we will define a family of adequacy criteria which is derived from the Rapps and Weyuker data flow testing criteria, and which circumvents this problem. A similar approach could be taken to deriving applicable criteria from the other structured testing criteria.

5. Thesis Overview

In this thesis we examine two families of test data adequacy criteria. For each of these families, we define the criteria for programs written in a large subset of Pascal, we investigate formal properties of the criteria, and we discuss software testing tools based on the criteria.

The first of these families, the *data flow testing criteria*, is derived from the family of criteria defined by Rapps and Weyuker. These criteria, which we define in section 1 of chapter 2, are based purely on the syntax of the program being tested. They require the test data to execute certain paths from points in the program at which a variables are defined to points where those definitions are used. The criteria are defined in such a way that the relationship among them is the same as the relationship among Rapps and Weyuker's family of criteria.

Weyuker has defined eight axioms which, she argues, should be satisfied by a "good" adequacy criterion [WEY86]. In Section 2.2, we explore some of the properties of the data flow testing criteria by examining which of the criteria satisfy which of Weyuker's axioms. We show that the criteria have some serious shortcomings, including failure to satisfy the applicability property.

In chapter three, we describe the design and implementation of a tool, ASSET, which is based on the data flow testing criteria. ASSET takes as input a program P, a test set T, and one of the data flow testing criteria, C, and determines whether T is adequate according to C for testing P.

The second family of criteria, the *feasible data flow testing criteria*, is defined in chapter 4, section 1. These criteria, which are derived from the data flow testing criteria, take into account some semantic information about the program being tested. The criteria are defined in such a way as to insure that they satisfy the applicability property. We show that that the relationship among the family of feasible data flow testing criteria differs significantly from the relationship among the family of data flow testing criteria. In section 4.2 we show that most of the feasible data flow testing criteria satisfy all of Weyuker's axioms.

Unfortunately, for any feasible data flow testing criterion C there is no algorithm to determine whether a given set of test data is C-adequate for testing a given program. The problem arises because in order to determine whether T is C-adequate, it may be necessary to determine whether some definition-use association can ever be exercised. That is, it may be necessary to determine whether there is an *executable* path, along which some variable v is not redefined, from a node d to a node u.

In chapter 5 we present a heuristic which attempts to determine whether any test data exists which exercises a given definition-use association. The heuristic is based on a combination of symbolic evaluation and data flow information. This heuristic, along with ASSET form the basis for an interactive feasible data flow testing tool.

CHAPTER 2:

The Data Flow Testing Criteria

1. Definitions of the Data Flow Testing Criteria

A family of test data adequacy criteria, based on analyzing the data flow characteristics of the program being tested, was defined in [RAP82,RAP85]. These criteria, which we call *data flow testing criteria*, or DF testing for short, were originally defined for a very simple universal programming language consisting of assignment statements, conditional and unconditional transfer statements, and I/O statements. They require that the test data exercise certain paths from a point in a program where a variable is defined to points where the variable is subsequently used. A tool, ASSET, which performs DF testing on programs written in such a language is described in [FRA85a].

In order to make data flow testing more practical, we have extended it to apply to a large subset of Pascal, and have enhanced ASSET accordingly. The basic ideas behind data flow testing apply to testing programs written in other imperative languages, but for precision it is necessary to specify a particular syntax. We now present the extended theory of data flow testing.

We apply DF testing to an individual subprogram, i.e., a main program, a procedure, or a function. To execute a procedure or function P, we must call it from a *driver program*. Thus, to test a procedure or function P we need a *test-set driver-program* pair (T,D), where D is a program which might call P and T is a subset of the input domain of the specification for D. Obviously, the path (or paths) through P which is executed when a particular test case is input to D will depend on D, as well as on the test case. We will often omit reference to the driver program, when it is obvious which driver program is calling the subprogram. Similarly, we may omit reference to the driver program if it simply reads in the arguments to the subprogram in order, then calls the subprogram once.

As a technical convenience we assume that the subprogram being tested has no **goto** statements, no **with** statements, no variant records, no functions having **var** parameters, no procedural or functional parameters, and no conformant arrays. It would not be difficult to relax these assumptions. We also assume that in every conditional statement, the Boolean expression which determines the flow of control has at least one occurrence of a variable or a call to the function *eof* or to the function *eoln*.

A subprogram can be uniquely decomposed into a set of disjoint blocks of statements. A *block* is a maximal sequence of simple statements having the properties that it can only be entered through the first statement and that whenever the first statement is executed, the remaining statements are executed in the given order. The subprogram to be tested is represented by a *flow graph* in which the nodes correspond to the blocks of the subprogram and edges indicate possible flow of control between blocks. As a technical convenience, some nodes which correspond to empty sequences of statements may also be added to the flow graph. Figure 2.1.1 shows the subgraphs corresponding to statements in the language. The subprogram's flow graph is obtained by merging the exit node of each statement with the entry node of the following statement. An entry node preceding the first statement of the procedure and an exit node succeeding the last statement are added.

We will call a flow graph formed according to the rules in Figure 2.1.1 a *structured flow graph*. All of the flow graphs considered in this thesis will be structured flow graphs. The "extra" nodes, corresponding to empty sequences of statements, have been added as a technical convenience, in order to simplify some of the algorithms in Chapter

FIGURE 2.1.1

Control flow and data flow for statements in the language

Si denotes the subgraph corresponding to statement Si.

SIMPLE STATEMENTS

Assignment statement: v := *expr*;

Node i has c-uses of each variable in
expr followed by a definition of v.

Input Output statements:

read(v1,...,vn);
readln(v1,...,vn);
read(f,v1,...,vn);
readln(f,v1,...,vn);

Node i has definitions of v1,...,vn.
If the file variable f is present then node i
also has a c-use followed by a definition of f.

write(e1,...,en);
writeln(e1,...,en);
write(f,e1,...,en);
writeln(f,e1,...,en);

Node i has c-uses of each variable occurring in e1,...,en.
If the file variable f is present then node i
also has a definition followed by a c-use of f.

Procedure call: P(e1,...,en)

Node i has c-uses of each variable occuring in
the expressions e1,...,en
[illegible] defin[illegible] of each actual
parameter [illegible] corresponds to a **var** formal parameter.

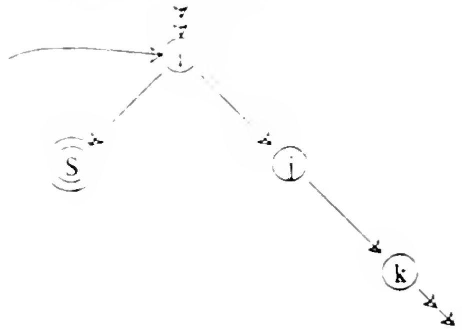
Nodes i and [illegible] are [illegible]ed to assure that
the procedure call has its own node.

continued on next page

REPETITIVE STATEMENTS

while statement: while B do S;

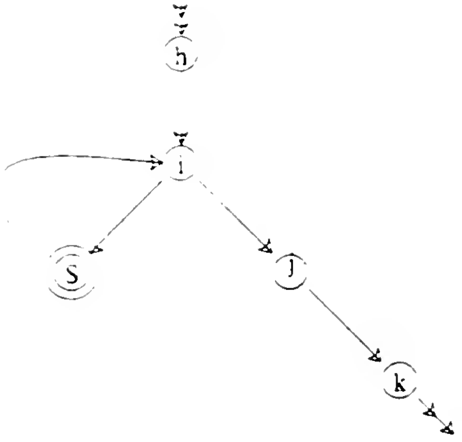
Let h be the entry node to subgraph S. Edges (i,h) and (i,j) have p-uses of each variable in the boolean expression B.



for statement:

for v:= e1 to e2 do S;
for v:= e1 downto e2 do S;

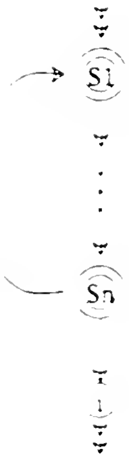
Let tmp be a new variable. Let f and g be the entry and exit nodes, respectively, of S. Node h has c-uses of each variable in e1, followed by a definition of v and c-uses of each variable in e2 followed by a definition of tmp. Edges (i,f) and (i,j) have p-uses of v and tmp. Node g has a c-use followed by a def of v.



repeat statement:

repeat S1;. ;Sn until B;

Let j be the entry node of S1, and let k be the exit node of Sn. Edges (k,j) and (k,i) have p-uses of each variable in the boolean expression B

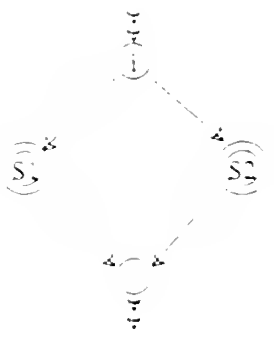


(continued on next page)

CONDITIONAL STATEMENTS

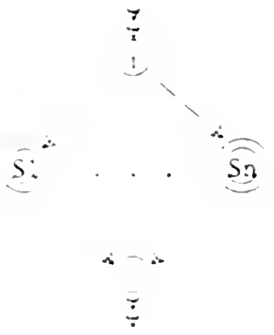
if-then-else statement

```
if B then S1
if B then S1 else S2;
```



Let k and j be the entry nodes of S1 and S2, respectively.
Edges (i,j) and (i,k) have p-uses of each variable in the boolean expression B
If there is no "else" part then subgraph S2 has a single node corresponding to an empty block.

```
case e1 of
   label-list1 : S1;
          :
          :
   label-listn : Sn
end
```



Let j1, ..., jn be the entry nodes of S1, ..., Sn, respectively.
Edges (i,j1), ..., (i,jn) have p-uses of each variable in the expression e1

5 and Appendix II.

Data flow analysis was originally used for compiler optimization [HEC77,SCH73]. It generally classifies each variable occurrence as being either a *definition*, in which a value is stored in a memory location, a *use*, in which a value is fetched from a memory location, or an *undefinition*, in which the value and the location become unbound. For our purposes, we will also distinguish between two different types of uses. The first type directly affects the computation being performed or outputs the result of some earlier definition. We call such a use a *computation use* or a *c-use*. Of course, a c-use may indirectly affect the flow of control through the subprogram. In contrast, the second type of use directly affects the flow of control through the subprogram, and thereby may indirectly affect the computations performed. We call such a use a *predicate use* or *p-use*.

We will associate a sequence of definitions and c-uses with each node in the flow graph and will associate a set of p-uses with each edge in the flow graph. Figure 2.1.1 shows the classification of variable occurrences in the language's statements. In addition, the entry node is considered to have a definition of each parameter, each non-local variable which occurs in the subprogram and of the input buffer *input*↑, which may implicitly occur in calls to the standard procedures/functions *read*, *readln*, *eoln* and *eof*. The exit node has an *undefinition* of each local variable, a c-use of each variable parameter, a c-use of each non-local variable, and a c-use of the input buffer, *input*↑.

We now discuss how data flow analysis is handled for structured variables. Since it is not in general possible to determine the particular array element which is being defined or used in an occurrence of an array variable, we will treat each entire array as a single entity. If a is an array variable, any definition of the variable $a[e]$ will consist of a c-use of each variable occurring in the expression e, followed by a definition of a. Any use of $a[e]$ will consist of uses of all of the variables occurring in e followed by a use of a.

Similarly, we will treat pointers purely syntactically, making no attempt to perform data flow analysis on dereferenced pointers. If p is a pointer variable, a definition of $p\uparrow$ consists of a c-use of p followed by a definition of $p\uparrow$, and a use of $p\uparrow$ consists of a use of p followed by a use of $p\uparrow$. Since it is not possible to statically determine the memory location to which a pointer points, we will ignore the definitions and uses of $p\uparrow$.

Each field of a record is treated as an individual variable. Any unqualified occurrence of a record is treated as an occurrence of each field of the record. Occurrences of file variables in I/O statements are handled by considering the effect of the statement on the file buffer.

Note that our model of data flow may not reflect the actual data flow in the subprogram being tested completely accurately. For example, we have made no attempt to perform any interprocedural data flow analysis, have ignored dereferenced pointers, have made no attempt to disambiguate array references, and have ignored potential aliasing and side effects. In an optimizing compiler it is imperative that conservative assumptions be made about the flow of data, lest a code transformation which changes the semantics of the program be performed. In the context of data flow testing, however, such caution is not strictly necessary. On the other hand, it seems reasonable to expect that more accurate data flow analysis will force the selection of better test data. In [FRA85b] we compare the test data needed to adequately test programs when each array is treated as a single entity to the test data required to adequately test transformed programs in which array references are disambiguated and each element of the array is treated as an individual entity. More exploration of the trade off between the difficulty of performing accurate data flow analysis and the quality of resulting test data is needed.

We are interested in tracing the flow of data *between* nodes, and thus define a c-use of a variable x in node i to be a *global c-use* if the value of x has been assigned in some block other than block i. Let x be a variable occurring in a subprogram. A path $(i,n_1,...,n_m,j)$, m≥0, containing no definitions or undefinitions of x in nodes $n_1,...,n_m$ is called a *definition clear path with respect to x* (def-clear path wrt x) from node i to node j and from node i to edge (n_m,j). A node i has a *global definition* of a variable x if it has a definition of x and there is a def-clear path wrt x from node i to some node containing a global c-use or edge containing a p-use of x. Since every p-use is associated with a potential transfer of control from one node to another, there is no need to distinguish between p-uses and global p-uses.

We restrict the class of subprograms to which data flow testing applies to those subprograms P satisfying the following two properties:

1. No-Syntactic-Undefined-P-use Property (NSUP):

   For every p-use of a variable x on an edge (i,j) in P, there is some path from the start node to edge (i,j) which contains a global definition of x.

2. Non-Straight-Line Property (NSL):

   P has at least one conditional or repetitive statement.

Note that the NSL property guarantees that at least one node in P's flow graph has more than one successor, and that at least one variable has a p-use in P.

The subprogram's *def-use graph* is obtained from the flow graph by associating with each node i, the sets *c-use(i)*={variables which have global c-uses in block i} and *def(i)*={variables which have global definitions in block i} and associating with each edge (i,j) the set *p-use(i,j)*={variables which have p-uses on edge (i,j)}. We also define sets of nodes *dcu(x,i)*={nodes j such that x∈c-use(j) and there is a def-clear path with respect to x from i to j} and *dpu(x,i)*={edges (j,k) such that x∈p-use(j,k) and there is a def-clear path with respect to x from i to (j,k)}. These definitions are summarized in Figure 2.1.2.

V	=	the set of variables
N	=	the set of nodes
E	=	the set of edges
def(i)	=	{x ∈ V | x has a global definition in block i}
c-use(i)	=	{x ∈ V | x has a global c-use in block i}
p-use(i,j)	=	{x ∈ V | x has a p-use in edge (i,j) }
dcu(x,i)	=	{j ∈ N | x ∈ c-use(j) and there is a def-clear path wrt x from i to j}
dpu(x,i)	=	{(j,k) ∈ E | x ∈ p-use(j,k) and there is a def-clear path wrt x from i to (j,k) }

Figure 2.1.2

Thus, if x∈def(i) and j∈dcu(x,i) then x has a global definition in node i and c-use in node j and there is a definition clear path with respect to x from node i to node j. Therefore it may be possible for control to reach node j with the variable x having the value which was assigned to it in node i.

A *definition-c-use association* is a triple (i,j,x) where i is a node containing a global definition of x and j∈ dcu(x,i). A *definition-p-use* association is a triple (i,(j,k),x) where i is a node containing a global definition of x and (j,k) ∈ dpu(x,i). A *simple path* is one in which all nodes, except possibly the first and last, are distinct. A *loop-free path* is one in which all nodes are distinct. A path $(n_1, \ldots, n_j, n_k)$ is a *du-path* with respect to a variable x if n_1 has a global definition of x and either

i) n_k has a global c-use of x and $(n_1, \ldots, n_j, n_k)$ is a def-clear simple path with respect to x, or

ii) (n_j, n_k) has a p-use of x and $(n_1, \ldots, n_j)$ is a def-clear loop-free path with respect to x.

An *association* is a definition-c-use association, a definition-p-use association, or a du-path.

Recall that a *complete path* is a path from the entry node to the exit node of the flow graph. A complete path π *covers* a definition-c-use association (i,j,x) [respectively a definition-p-use association (i,(j,k),x)] if it has a definition clear subpath with respect to x from i to j [respectively, from i to (j,k)]. π covers a du-path π' if π' is a subpath of π. A set Π of complete paths covers an association if some element of the set does. A test-set/driver-program pair (D,T) covers an association if when input to D the elements of T cause the execution of the set of paths Π, and Π covers the association.

Roughly speaking, the family of data flow testing criteria is based on requiring that the test data execute definition clear paths from each node containing a global definition of a variable to specified nodes containing global c-uses and edges containing p-uses of that variable. For each variable definition we can demand that $\begin{bmatrix} all \\ some \end{bmatrix}$ definition clear paths with respect to that variable from the node containing the definition to $\begin{bmatrix} all \\ some \end{bmatrix}$ of the $\begin{bmatrix} uses \\ c-uses \\ p-uses \end{bmatrix}$ reachable by some such path be executed. The criteria are defined precisely in Figure 2.1.3.

If variable x is has a global definition in node i, the all-defs criterion requires the test data to exercise *some* path which goes from node i to *some* node or edge at which the value assigned to x in node i is used. The all-uses criterion requires the test data to exercise *at least one* path to *each* such node or edge. The all-du-paths criterion requires that *all* of the du-paths from i to *each* such node or edge be exercised. The criteria all-p-uses,

THE DATA FLOW TESTING CRITERIA

A test-set/driver-program pair (T,D) satisfies criterion C for subprogram P if and only if for each node i in P's flow graph and each $x \in$ def(i) the set Π of paths executed by T covers the following associations:

CRITERION	ASSOCIATIONS REQUIRED
All-defs	Some (i,j,x) s.t. $j \in$ dcu(x,i) or some (i,(j,k),x) s.t. $(j,k) \in$ dpu(x,i).
All-c-uses	All (i,j,x) s.t. $j \in$ dcu(x,i).
All-p-uses	All (i,(j,k),x) s.t. $(j,k) \in$ dpu(x,i).
All-p-uses/some-c-uses	All (i,(j,k),x) s.t. $(j,k) \in$ dpu(x,i). In addition, if dpu(x,i)=$\emptyset$ then some (i,j,x) s.t. $j \in$ dcu(x,i). Note that since i has a *global* definition of x, dpu(x,i)=$\emptyset \Rightarrow$ dcu(x,i)$\neq \emptyset$.
All-c-uses/some-p-uses	All (i,j,x) s.t. $j \in$ dcu(x,i). In addition, if dcu(x,i)=$\emptyset$ then some (i,(j,k),x) s.t. $(j,k) \in$ dcu(x,i). Note that since i has a *global* definition of x, dcu(x,i)=$\emptyset \Rightarrow$ dpu(x,i)$\neq \emptyset$.
All-uses	All (i,j,x) s.t. $j \in$ dcu(x,i) and all (i,(j,k),x) s.t. $(j,k) \in$ dpu(x,i).
All-du-paths	All du-paths from i to j with respect to x for each $j \in$ dcu(x,i) and all du-paths from i to (j,k) with respect to x for each $(j,k) \in$ dpu(x,i).

For comparison we also define the criteria all-nodes (all-edges, all-paths, respectively) which require that Π cover every node (every edge, every path, respectively) in the flow graph

Figure 2.1.3

all-c-uses, all-p-uses/some-c-uses, and all-c-uses/some-p-uses place emphasis on either c-uses or p-uses. Note that any subprogram has only finitely many definition-use associations, so none of the DF criteria requires an infinite amount of test data. Upper bounds on the amount of test data required by the DF criteria are established in [WEY84].

Criterion C_1 *includes* criterion C_2 if and only if for every subprogram, any test-set/driver-program pair which satisfies C_1 also satisfies C_2. Criterion C_1 *strictly includes* criterion C_2, denoted $C_1 \Rightarrow C_2$, if and only if C_1 *includes* C_2 and for some subprogram P there is a test-set/driver-program pair which satisfies C_2 but does not satisfy C_1. The notion of *subsumption* in [CLA85] is similar to the notion of inclusion.

Rapps and Weyuker proved that for the simple language for which DF testing was originally defined, the relationship among the criteria is as shown in Figure 2.1.4 [RAP85]. Clarke et. al. [CLA86] have shown the relationship of the criteria defined by Laski and Korel [LAS83] and Ntafos [NTA84] to the DF criteria.

In extending the theory of DF testing to apply to programs written in Pascal we have preserved the inclusion relations among the DF criteria. Doing so required the inclusion of definitions of all non-local variables in the entry node of the procedure and careful treatment of implicit uses of the variable *input*↑. For symmetry, we have also added the all-c-uses criterion, which was not defined in [RAP85].

THEOREM 2.1.1:

The relationship between the data flow testing criteria is as shown in Figure 2.1.4. That is, DF criterion C1 includes criterion C2 if and only if the inclusion is explicitly shown in the figure, or if it follows from the transitivity of the relation.

PROOF:

$$\begin{tikzcd}
 & \text{ALL-PATHS} \arrow[d] & \\
 & \text{ALL-DU-PATHS} \arrow[d] & \\
 & \text{ALL-USES} \arrow[dl] \arrow[dr] & \\
\text{ALL-C-USES SOME-P-USES} \arrow[d] \arrow[dr] & & \text{ALL-P-USES SOME-C-USES} \arrow[dl] \arrow[d] \\
\text{ALL-C-USES} & \text{ALL-DEFS} & \text{ALL-P-USES} \arrow[d] \\
 & & \text{ALL-EDGES} \arrow[d] \\
 & & \text{ALL-NODES}
\end{tikzcd}$$

FIGURE 2.1.4

The relationship among the data flow testing criteria

Rapps and Weyuker proved a theorem analogous to this one for the data flow testing criteria as originally defined in [RAP85]. Some minor modification of their proof is needed to account for the fact that our flow graphs differ from those used by Rapps and Weyuker. For symmetry, we have also added the all-c-uses criterion, which was omitted from the original family of criteria.

The proofs of all of the inclusions except (all-p-uses includes all-edges) are obvious and will be omitted. The proofs of strictness of inclusions and of incomparability are essentially the same as the corresponding proof in [RAP85]. The same *programs* and *test sets* used in that proof can be used here. Although there are some minor differences in the form of the flow graphs corresponding to these programs, the proofs go through.

It remains to show that all-p-uses $\Rightarrow$ all-edges, that all-c-uses/some-p-uses $\Rightarrow$ all-c-uses, and that all-c-uses is incomparable with the other criteria.

Proof that all-p-uses $\Rightarrow$ all-edges:

Let T be a test set which satisfies all-p-uses for P and let Π be the set of paths through P's flow graph which are executed by T. Let (i,j) be an edge in the flow graph.

case 1: (i,j) is labelled.

Then since P satisfies the NSUP property, there is a definition-use association (k,(i,j),x) for some variable x and node k. Since T satisfies all-p-uses, Π covers (k,(i,j),x), so, Π covers edge (i,j).

case 2: (i,j) is not labelled.

case 2a: There only one path from the entry node to edge (i,j).

Then every path in Π covers edge (i,j). Since P satisfies NSUP and NSL, Π is not empty, so there is at least one such path. Thus P covers edge (i,j).

case 2b: There is more than one path from the entry node to edge (i,j).

Let n be the unique node such that

1) n has more than one successor, and

2) there exists exactly one acyclic path from node n to edge (i,j).

Let m be the unique successor of n such that there is an acyclic path from n to m to (i,j). Then every path which covers edge (n,m) covers edge (i,j). Since n has more than one successor, edge (n,m) is labelled, so (by NSUP), there is a def-p-use association (k,(n,m),x). Π covers (k,(n,m),x), hence covers (n,m), hence covers (i,j).

Proof that all-c-uses/some-p-uses $\Rightarrow$ all-c-uses.

It is obvious that all-c-uses/some-p-uses includes all-c-uses. To see that the inclusion is strict, consider the program whose flow graph is shown in Figure 2.1.5. The only definition-c-use associations in this program are (4,8,x) and (1,10,input$\uparrow$). The set T={(0,0)} satisfies all-c-uses for this program, but it does not cover either of the definition-p-use associations (3,(6,7),z) or (3,(6,8),z), hence does not satisfy all-c-uses/some-p-uses.

Proof that all-c-uses is incomparable with the other criteria:

The set T={(0,0)} satisfies all-c-uses for the program shown in Figure 2.1.5, but does not satisfy all-p-uses/some-c-uses, all-p-uses, all-edges, all-nodes, or all-defs, so all-c-uses does not include any of these criteria.

To see that none of the criteria all-p-uses/some-c-uses, all-p-uses, all-edges, all-nodes, or all-defs include the all-c-uses criterion, consider the program shown in Figure 2.1.6. The test set {(-1),(5)} satisfies the criteria all-p-uses/some-c-uses, all-p-uses, all-edges, all-nodes, and all-defs. However, it does not cover the def-c-use association

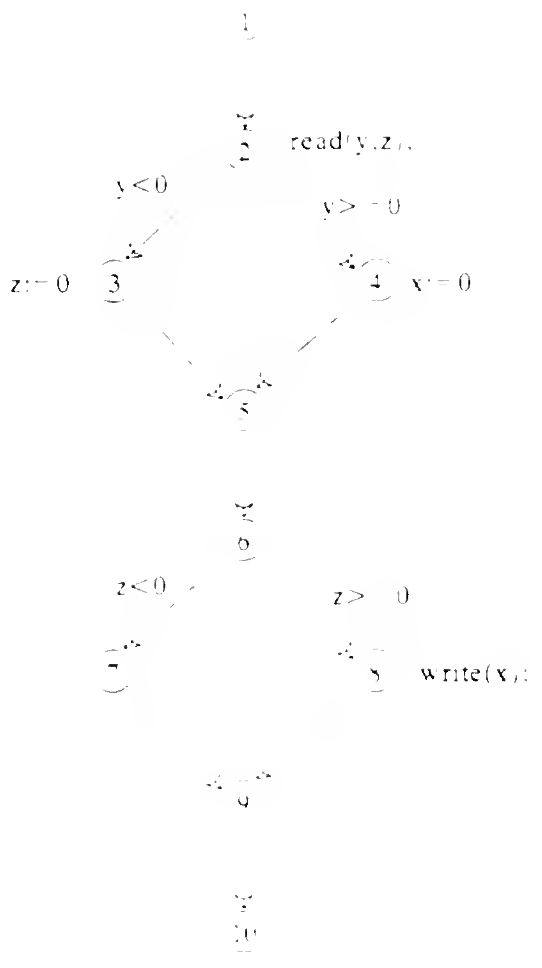


FIGURE 2.1.5

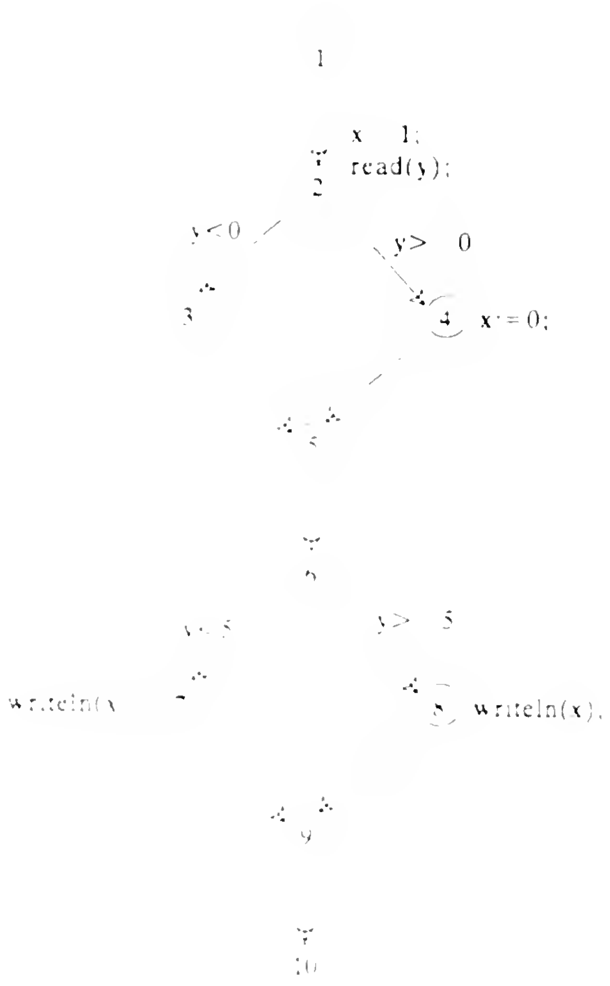


FIGURE 2.1.6

(4,7,x), and hence does not satisfy the all-c-uses criterion.■

2. Axiomatic Evaluation of Data Flow Testing Criteria

Weyuker has defined eight axioms which, she argues, should be satisfied by a "good" adequacy criterion [WEY86]. In this section we explore which of these axioms are satisfied by the data flow testing criteria.

Weyuker's model of computation assumes that the objects being tested are single-entry, single-exit *programs* which take all of their inputs at the beginning and which have a single output statement at the end. We will make certain minor modifications in the axioms in order to make them apply to testing Pascal *subprograms*.

Axiom 1: Applicability

For every subprogram, there exists an adequate test set.

None of the data flow testing criteria satisfy the applicability axiom. Consider the following program, whose flow graph is shown in Figure 2.2.1.

```
program Applicability_CounterExample;
var x,y,z: integer;
begin
        read(x,y,z);
        if x ≥ 0 then
        begin
                y := 0;
                z := 0;
        end;
        if x < 0 then
        begin
                writeln(y);
                if z≥0 then writeln('hello')
        end,
end.
```

There are no executable paths which cover the association (3,8,y), which is required by the all-defs and all-c-uses criteria, nor the associations (3,(8,9),z), or (3,(9,10),z),

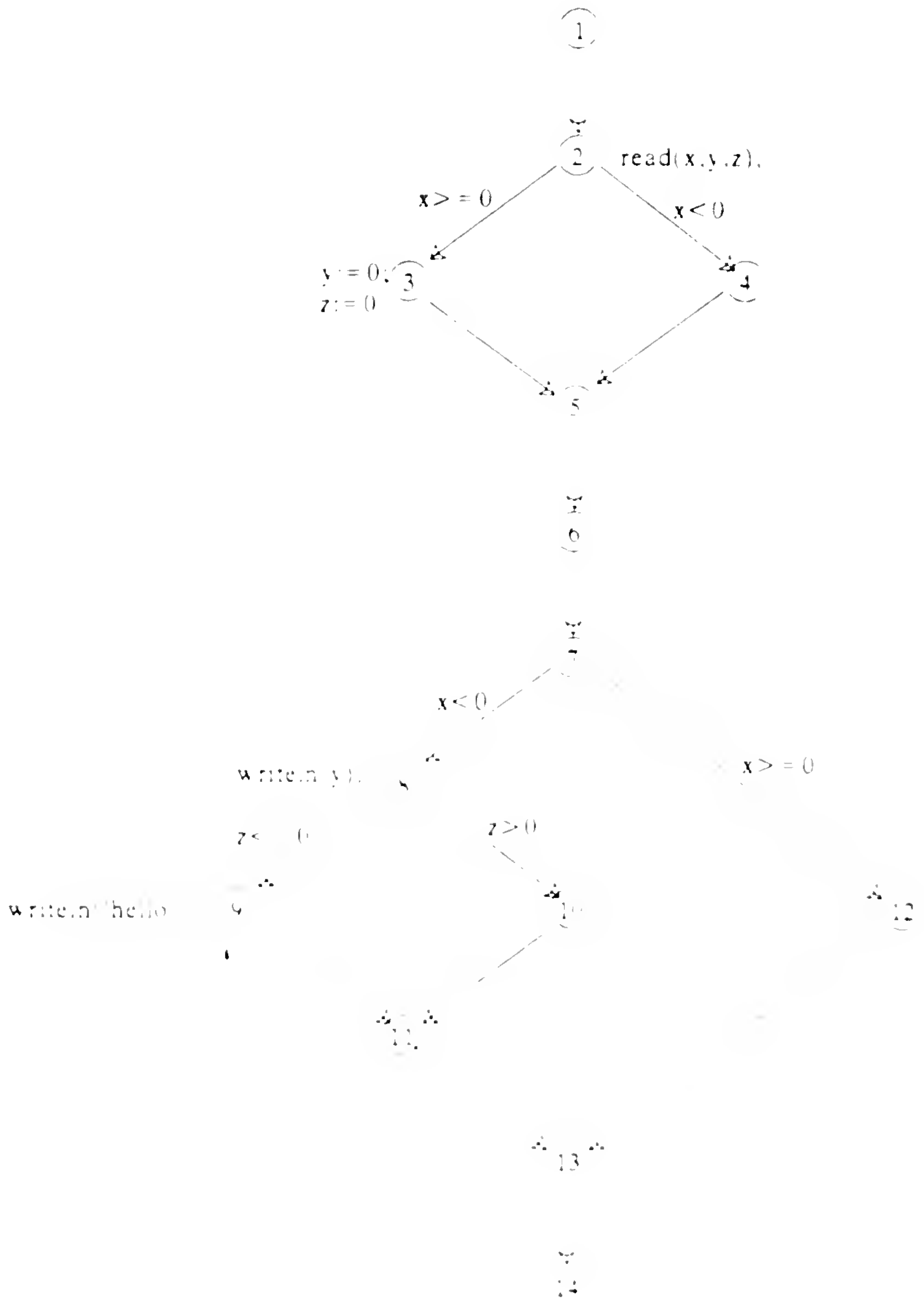

1
2
read(x,y,z);
x > = 0
x < 0
y := 0;
z := 0
3
4
5
6
7
x < 0
x > = 0
writeln(y);
8
z > 0
writeln('hello'
9
10
12
11
13
14

FIGURE 2.2.1

which are required by the all p-uses criterion. Therefore no test set is C-adequate for testing this program, for any data flow criterion C. Note that this program does not have any unexecutable code.

While the above program was contrived to demonstrate that *none* of the criteria satisfy the applicability axiom, there are many "realistic" subprograms for which no set of test data is C-adequate when C is all-p-uses, all-c-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, all-uses, or all-du-paths. For example, any subprogram having a **for** loop in which the upper bound is always greater than or equal to the lower bound has a def-p-use association which is not covered by any executable path. In Figure 2.2.2, the only executable path from the definition of i in node 1 to the p-use of i on edge (2,4) is the path which goes around the loop 5 times. Since this path has a definition of i, we can conclude that no executable path covers the definition-p-use association (1,(2,4),i).

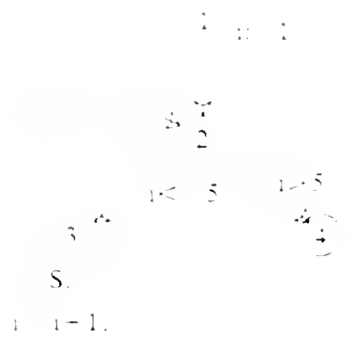
Axiom 2: Non-Exhaustive Applicability

There is a subprogram P and test set T such that P is adequately tested by T and T is not an exhaustive test set.

All of the data flow testing criteria satisfy this axiom. Consider the following program whose flow graph is shown in Figure 2.2.3.

```
program Non-exhaustive_Applicability(input,output);
var x : integer;
begin
        read(x);
        if x>0 then writeln('hello')
        else writeln('goodbye')
end
```

The test set T={(0),(1)} is C-adequate for testing this program, where C is any data flow testing criterion. On the other hand, the exhaustive test set for this program is the set of



Subgraph corresponding to the statement
for i:=1 to 5 do S

FIGURE 2.2.2

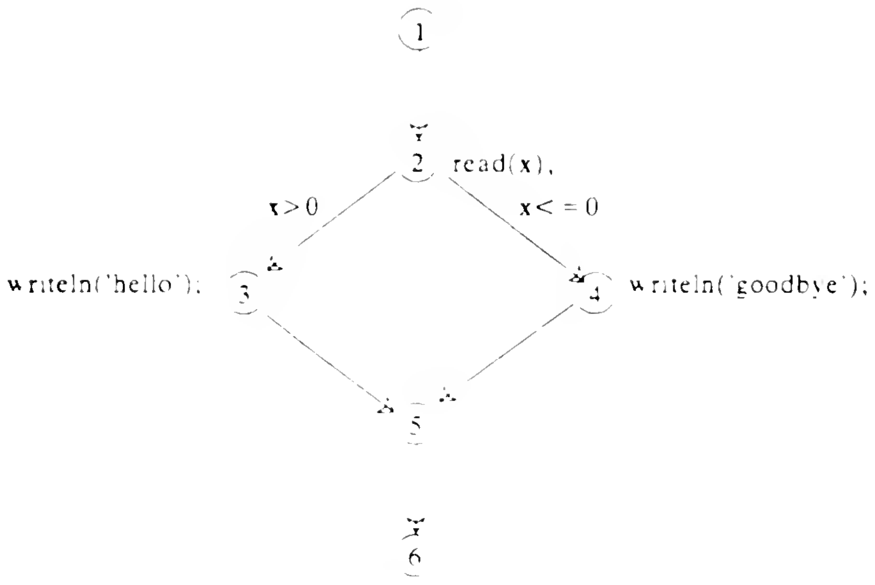


FIGURE 2.2.3

integers.

Axiom 3: Monotonicity

If T is adequate for P and T$\subseteq T'$ then T' is adequate for P.

It is obvious that all of the data flow criteria satisfy the monotonicity property since all of the associations covered by T are also covered by T'.

Axiom 4: Inadequate Empty Set

The empty set is not adequate for any subprogram.

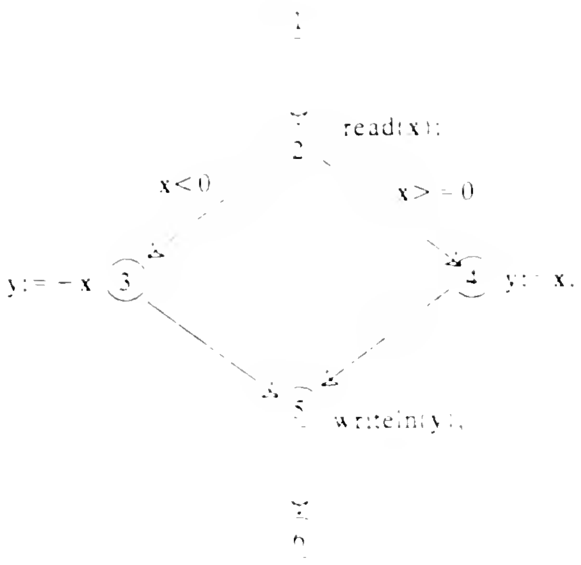
Since every subprogram in the class we are considering satisfies the No-Syntactic-Undefined-P-uses Property and the Non-straight-line Property, each subprogram has at least one definition-p-use association. It follows that the empty set is not adequate for any of the criteria all-p-uses, all-p-uses/some-c-uses, all-c-uses/some-p-uses, all-uses, or all-du-paths.

Since every subprogram has a definition of *input*↑ in the entry node and a c-use of *input*↑ in the exit node, the all-c-uses criterion satisfies the inadequate empty set axiom. Note, however, that the fact that all-c-uses satisfies this axiom can be construed as being an artifact of the definitions.

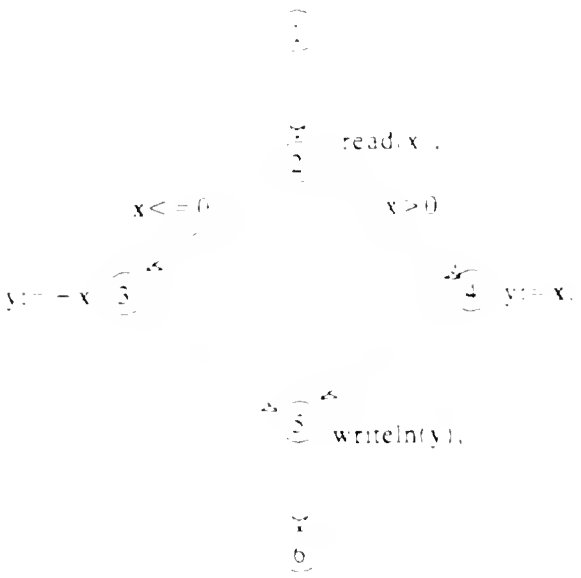
Axiom 5: Anti-extensionality

There are subprograms P and Q such that P is equivalent to Q, T is adequate for P, but T is not adequate for Q.

All of the data flow testing criteria satisfy the Anti-extensionality property. Consider the programs P and Q whose flow graphs are shown in Figure 2.2.1. Both of these programs take as input a single integer, x, and output the absolute value of x. The test set



FLOWGRAPH FOR PROGRAM P



FLOWGRAPH FOR PROGRAM Q

FIGURE 2.2.4

T={(−1),(0)} is C-adequate for P for any data flow testing criterion C. However, since T does not cover node 3 in Q, it does not satisfy the all-nodes criterion, hence does not satisfy all-p-uses, all-p-uses/some-c-uses, all-uses, or all-du-paths for Q. T does not cover the def-c-use association (4,5,y) in Q, which is required by the all-defs criterion, and hence does not satisfy the criteria all-defs, all-c-uses, all-c-uses/some-p-uses for Q. So while T satisfies all of the criteria for P, it satisfies none of them for the equivalent program Q.

Axiom 6: General Multiple Change

There are subprograms P and Q which are the "same shape", and a test set T such that T is adequate for P, but T is not adequate for Q. (Roughly speaking, P and Q are the same shape if P can be transformed into Q by performing one or more simple transformations such as replacing a relational operator with another relational operator, replacing a constant with another constant, etc.)

All of the data flow testing criteria satisfy this axiom. Consider again the programs in Figure 2.2.4. These programs are the same shape because Q is obtained from P by changing the relational operator in an if-then-else statement. As shown above, The test set T={(−1),(0)} satisfies all of the data flow testing criteria for P, but satisfies none of the criteria for Q. Note that when considering the anti-extensionality axiom, the relevant feature of programs P and Q was their semantic closeness, but when considering the general multiple change axiom, the relevant feature is their syntactic closeness.

Axiom 7: Antidecomposition

There exists a subprogram P and a "component" Q such that T is adequate for P, *T'* is the set of vectors of values that that variables can assume on entrance to Q for some t of T.

and T' is not adequate for Q.

A *component* of P is a contiguous sequence of statements of P. Since our definitions of the data flow testing criteria apply to testing a *subprogram*, not a component, we will modify the axiom slightly. The modified axiom captures the intuition behind Weyuker's axiom, but fits our model of computation.

Antidecomposition:

There is a subprogram P which calls a subprogram Q and a test T such that T is adequate for the program P' obtained from P by expanding in-line one call to Q, but the test-set/driver-program pair (T,P) is not adequate for Q.

None of the data flow testing criteria except all-c-uses and all-c-uses/some-p-uses satisfy the antidecomposition axiom. The fact that all-c-uses and all-c-uses/some-p-uses do satisfy the axiom is an artifact of the definitions, which we will discuss below.

With certain exceptions which we will examine below, each definition-use association a in Q corresponds to a definition-use association a' in the portion of P' which corresponds to Q. The test case which covers a' when input to P' will also cover a when P is run.

For example, consider the following programs, whose flow graphs are shown in Figure 2.2.5.

```
program P(input,output);
var x : integer;
procedure Q(x: integer);
begin
      if x≥0 then writeln('hello')
      else writeln('goodbye')
end; {Q}
begin {P}
      read(x);
      Q(x);
```

```
        write(x)
end.


program P'(input,output);
var x,y,t1 : integer;
begin {P}
        read(x);

        {in-line expansion of call to Q begins}
        t1 := x;
        if t1≥0 then writeln('hello')
        else writeln('goodbye')
        {in-line expansion of call to Q ends}

        write(x)
end.
```

The association (6,(7,8),x) in Q, which arises from the fact that the parameter x has a definition in node 1 and a p-use on edge(7,8), corresponds to the association (13,(13,14),t1) in *P'*. Both are covered by any test case consisting of an integer which is greater than or equal to 0.

The exceptions arise due to the fact that we have given the exit node of a procedure a c-use of each **var** parameter and each global variable. If some global variable (or **var** parameter) has a definition in Q which reaches the exit node of Q and if that variable (or the actual parameter corresponding to the **var** formal parameter) is never used at any point in P which is reachable from the call to Q, then Q will have a definition-c-use association which does not correspond to any definition-c-use association in *P'*.

For example, consider the programs whose flow graphs are shown in Figure 2.2.6. Let y be a global variable. Procedure Q has a definition of y in node 9 which reaches the exit node of Q and which is never used in the calling program, P. The test set T={(0)} is adequate for *P'* according to the all-defs, all-c-uses, and all-c-uses/some-p-uses criterion. However, T is not adequate for Q according to any of these criteria. Because it is non-

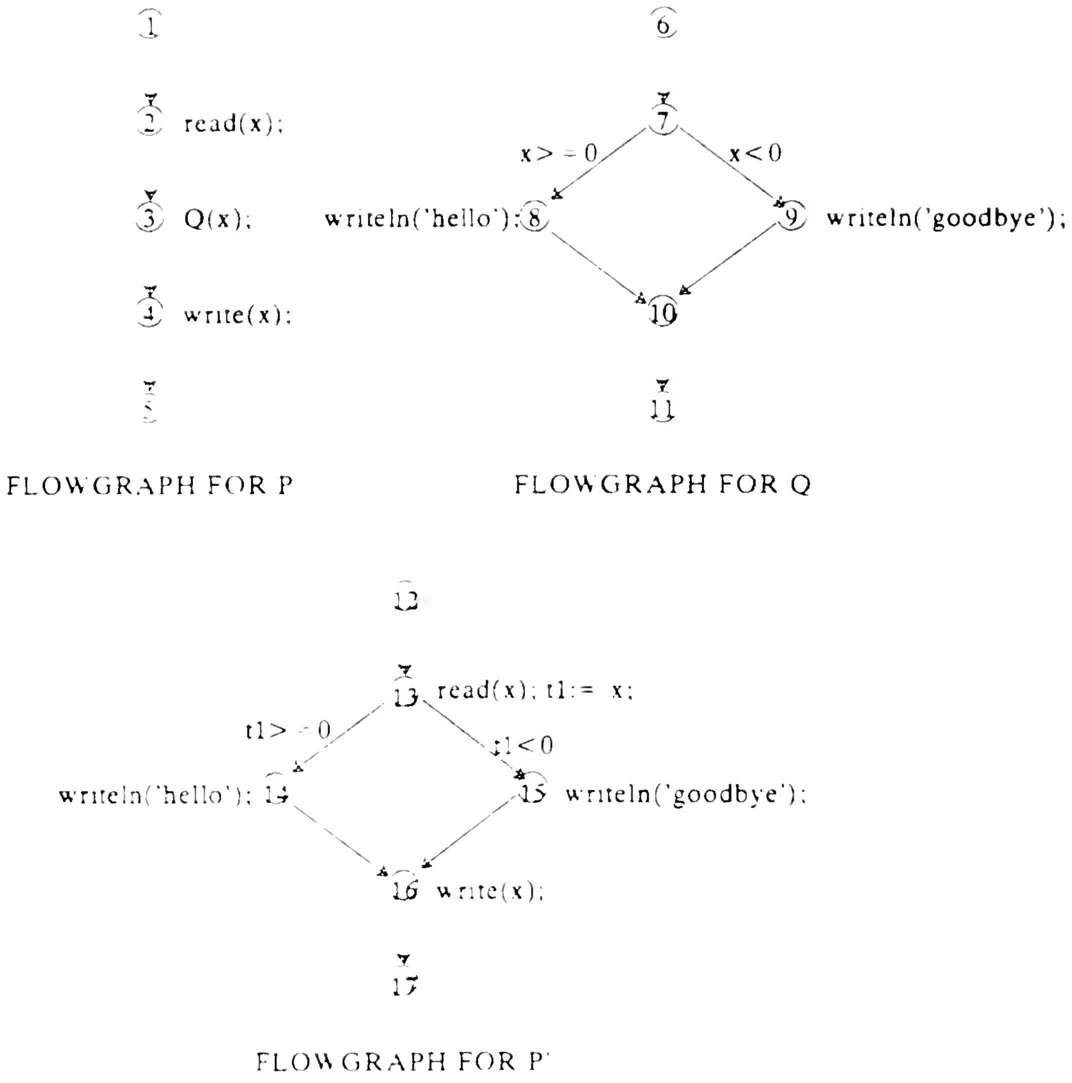
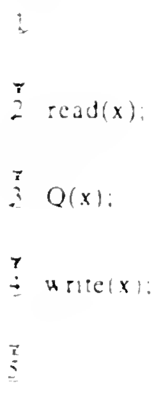
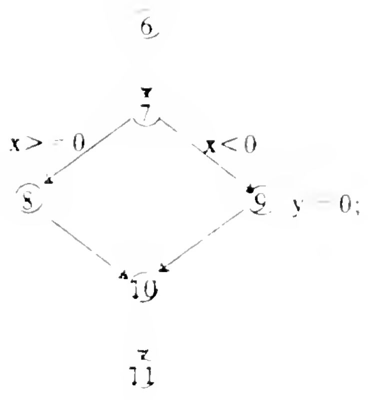

read(x);
Q(x);
write(x);
FLOWGRAPH FOR P
x> = 0
x<0
writeln('hello');
writeln('goodbye');
FLOWGRAPH FOR Q
read(x); t1:= x;
t1> = 0
t1<0
writeln('hello');
writeln('goodbye');
write(x);
FLOWGRAPH FOR P'

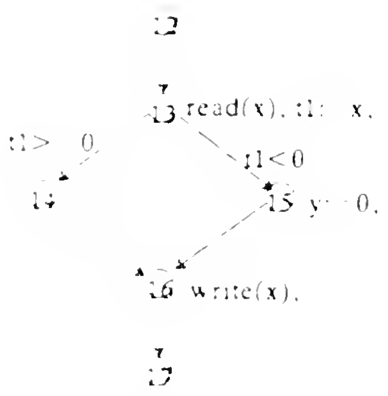
FIGURE 2.2.5



FLOWGRAPH FOR P



FLOWGRAPH FOR Q



FLOWGRAPH FOR P'

FIGURE 2.2.6

local to Q, y has a c-use in node 11, the exit node of Q. To satisfy all-defs, all-c-uses, or all-c-uses/some-p-uses for Q a test set must cover the definition c-use association (9,11,y), thus must include an element (Y), where Y is strictly less than zero.

Note that the fact that these criteria satisfy the antidecomposition axiom is an artifact of the definitions. If the model of data flow were based on accurate interprocedural data flow analysis, or if the class of subprograms were restricted to exclude those having data flow anomalies like the one in the program in Figure 2.2.6 then the criteria would fail to satisfy the axiom.

Axiom 8: Anticomposition

There exist subprograms P and Q and test set T such that T is adequate for P and P(T) is adequate for Q, but T is not adequate for P;Q.

Weyuker's model of computation assumes that all input statements appear at the beginning of a program, and that all output statements appear at the end of a program. "P;Q" denotes the program obtained from two programs P and Q, which use the same set of identifiers, by replacing P's unique output and exit statements by Q' where Q' is obtained from Q by deleting the input statements. We will modify the Anticomposition axiom slightly obtain an axiom which captures the intuition motivating Weyuker's definition, but which fits our model of computation.

Anticomposition:

There is a subprogram R whose statement part consists of a call to procedure P followed immediately by a call to procedure Q, and a test set T for R such that the test-set/driver-program pair (T,R) is adequate for P and for Q, but T is not adequate for the subprogram R' obtained from R by expanding P and Q in-line.

All of the data flow testing criteria satisfy the Anticomposition axiom. To see that the criteria all-defs, all-c-uses, all-c-uses/some-p-uses, all-uses, and all-du-paths satisfy the axiom, consider the following programs, whose flow graphs are shown in Figure 2.2.7

```
program R(input,output);
var x,y,z: integer;
procedure P;
        begin {P}
                y:=0;
                read(x);
                if x >= 0 then y:= 1;
        end; {P}
procedure Q;
        begin {Q}
                read(z);
                if z < 0 then writeln(y);
        end; {P}
begin {R}
        P;
        Q
end.
```

```
program R'(input,output);
var x,y,z: integer;
begin {R'}
        y:=0;
        read(x);
        if x >= 0 then y:= 1;
        read(z);
        if z < 0 then writeln(y);
end.
```

Let T={(−1,−1),(0,0)}. Then the test set/driver program pair (T,R) satisfies the all of the data flow testing criteria for P and for Q. However, T does not satisfy the criteria all-defs, all-c-uses, all-c-uses/some-p-uses, all-uses, or all-du-paths for R'. To do so, T would have to cover the definition-c-use association (16,19,y). That is, T would have to include an element (a,b) where a≥0 and b<0. A similar example shows that the criteria all-p-uses and all-p-uses/some-c-uses also satisfy the anticomposition axiom.

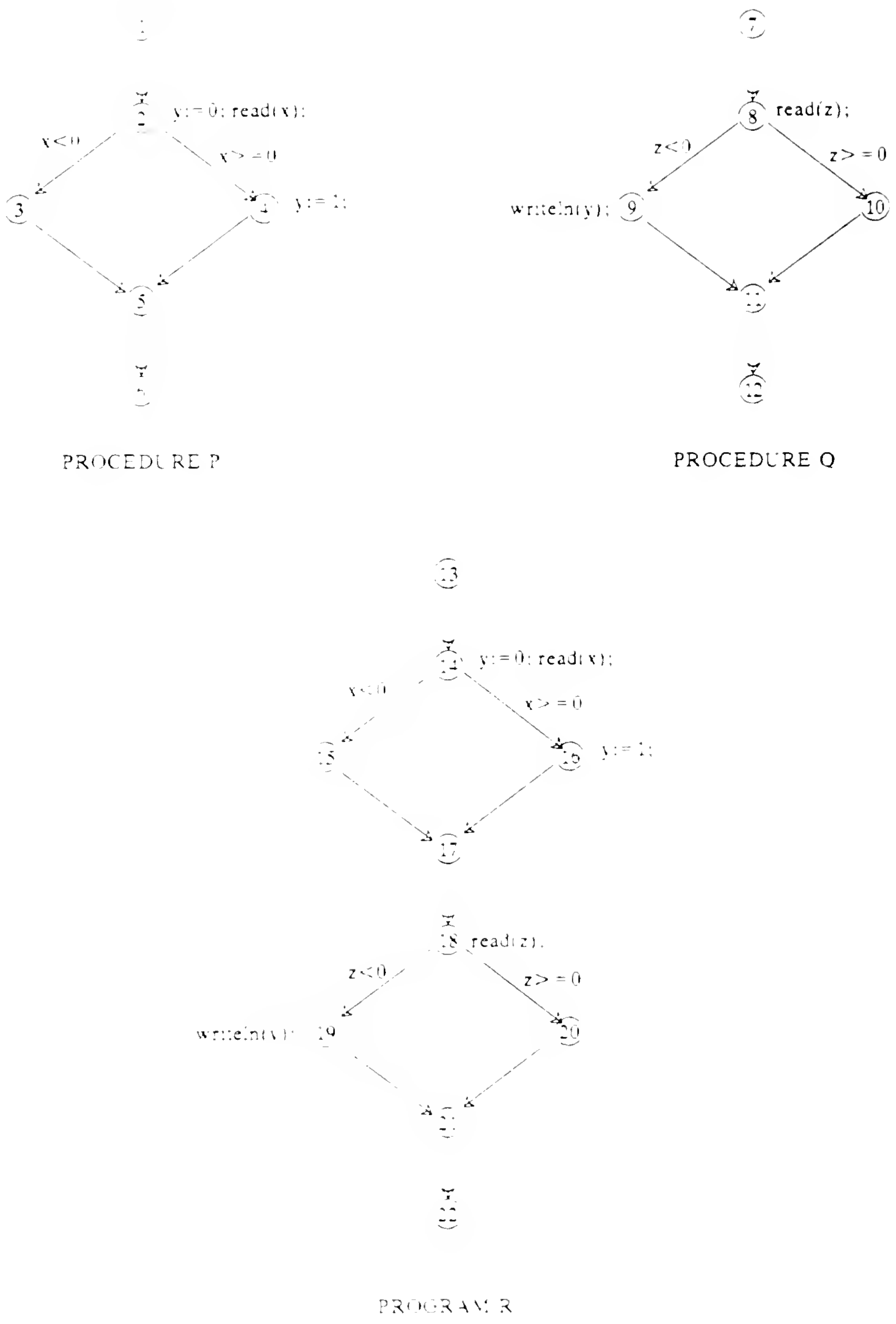

y:=0; read(x);
x<0
x>=0
y:=1;
PROCEDURE P
read(z);
z<0
z>=0
writeln(y);
PROCEDURE Q
y:=0; read(x);
x<0
x>=0
y:=1;
read(z);
z<0
z>=0
writeln(y);
PROGRAM R

FIGURE 2.27

Summary

The above results are summarized in the table shown in Figure 2.2.8.

	Axioms							
Criteria	1	2	3	4	5	6	7	8
all-defs	no	yes	yes	yes	yes	yes	no	yes
all-c-uses	no	yes	yes	yes	yes	yes	yes	yes
all-p-uses	no	yes	yes	yes	yes	yes	no	yes
all-c-uses/ some-p-uses	no	yes	yes	yes	yes	yes	yes	yes
all-p-uses/ some-c-uses	no	yes	yes	yes	yes	yes	no	yes
all-uses	no	yes	yes	yes	yes	yes	no	yes
all-du-paths	no	yes	yes	yes	yes	yes	no	yes

FIGURE 2.2.8
Summary of Axiomatic Evaluation of the Data Flow Testing Criteria

The fact that none of the data flow testing criteria satisfy the applicability axiom is disappointing, though hardly surprising. Statement testing, branch testing, and path testing also fail to satisfy the applicability axiom, for essentially the same reason. All of these criteria base the choice of paths which the test data must execute solely on the syntax of the subprogram being tested. If some required path is unexecutable then the subprogram cannot be adequately tested.

One could argue that the subprograms which cannot be adequately tested according to the statement testing or branch testing criteria are in some sense pathological because they have unexecutable statements or unexecutable branches. (On the other hand, one could argue that there are good reasons, arising from "defensive programming" for programs to have unexecutable code.) Be that as it may, the situation is somewhat different for the data flow testing criteria. There are programs which have no unexecutable statements or branches yet which cannot be adequately tested by the data flow testing criteria

Practitioners generally dodge the issue of inapplicable criteria by eliminating from consideration those statements, branches, or def-use associations which cannot be executed. Since it is undecidable whether a given statement, branch or def-use association can be executed, this process cannot be fully automated.

In chapter four we will define a new family of adequacy criteria which are derived from the data flow testing criteria by eliminating unexecutable associations from consideration. These new criteria do satisfy the applicability axiom. We will see that there are some significant differences between the formal properties of this family and those of the data flow testing criteria.

CHAPTER 3:

The Design and Implementation of ASSET

1. Introduction

We have built a software testing tool called ASSET (A System to Select and Evaluate Tests) which is based on the data flow testing criteria. In this chapter we will discuss ASSET's design and implementation.

ASSET takes as input a Pascal program P, the name of a subprogram Q of P, a set of test data, and one of the data flow testing criteria. It runs the program P on the test data and produces a list of those definition-use associations in Q which are required by the given criterion but not covered by the test data.

The user then examines the output to determine whether the program has behaved according to its specification. If the program computed the wrong output, the user reports that the program has a error. If the program has behaved correctly on all of the test cases, and if the criterion has not been satisfied, the user selects additional test data and continues the testing process. If the program has behaved correctly and the criterion has been satisfied the user can either release the program, certifying that the subprogram Q has been adequately tested according to the criterion, or can select another criterion and continue.

In section 2 we describe ASSET's design and in section 3 we describe the implementation of the current version of ASSET, which is an experimental prototype. A script of an ASSET session can be found in section 6 of the ASSET User manual, which is included as Appendix 1.

2. The Design of ASSET

ASSET is an interactive system consisting (primarily) of five modules which communicate with each other via files. Execution of the modules is invoked by commands issued by the user, via a user interface. The modules are (A) Front End, (B) Association-Finder, (C) Compiler, (D) Tester, and (E) Checker. The front end is the only module which is concerned with the syntax of the language in which the program being tested is written. Thus, extending ASSET to handle programs written in other languages should be fairly straight forward.

In this section we describe each of these modules and their interaction with one another. Figure 3.2.1 is a schematic diagram of ASSET's design. We conclude this section with a discussion of some of the limitations of this design.

Module A: The Front End

This module is similar to the front end of a compiler. It takes as input a syntactically correct program (written in the subset of Pascal described in Chapter 2) P, which we will call the *subject program*, and a subprogram Q of P, which we will call the *subject procedure*. It produces a symbol table with information on the identifiers visible to the statement part of Q, the flow graph of Q, a table called the *def-use* table, and a program called the *modified subject program*. The def-use table lists the definitions and c-uses in each node and p-uses on each edge. The modified subject program is a program which, for each test case t, simulates the behavior of P on input t, and which also produces a trace of the path through Q's flow graph which is executed when P is run with input t.

To do this, the front end must perform lexical analysis, parse all declarations which are visible to the statement part of Q, and (at least partially) parse the statement part of Q.

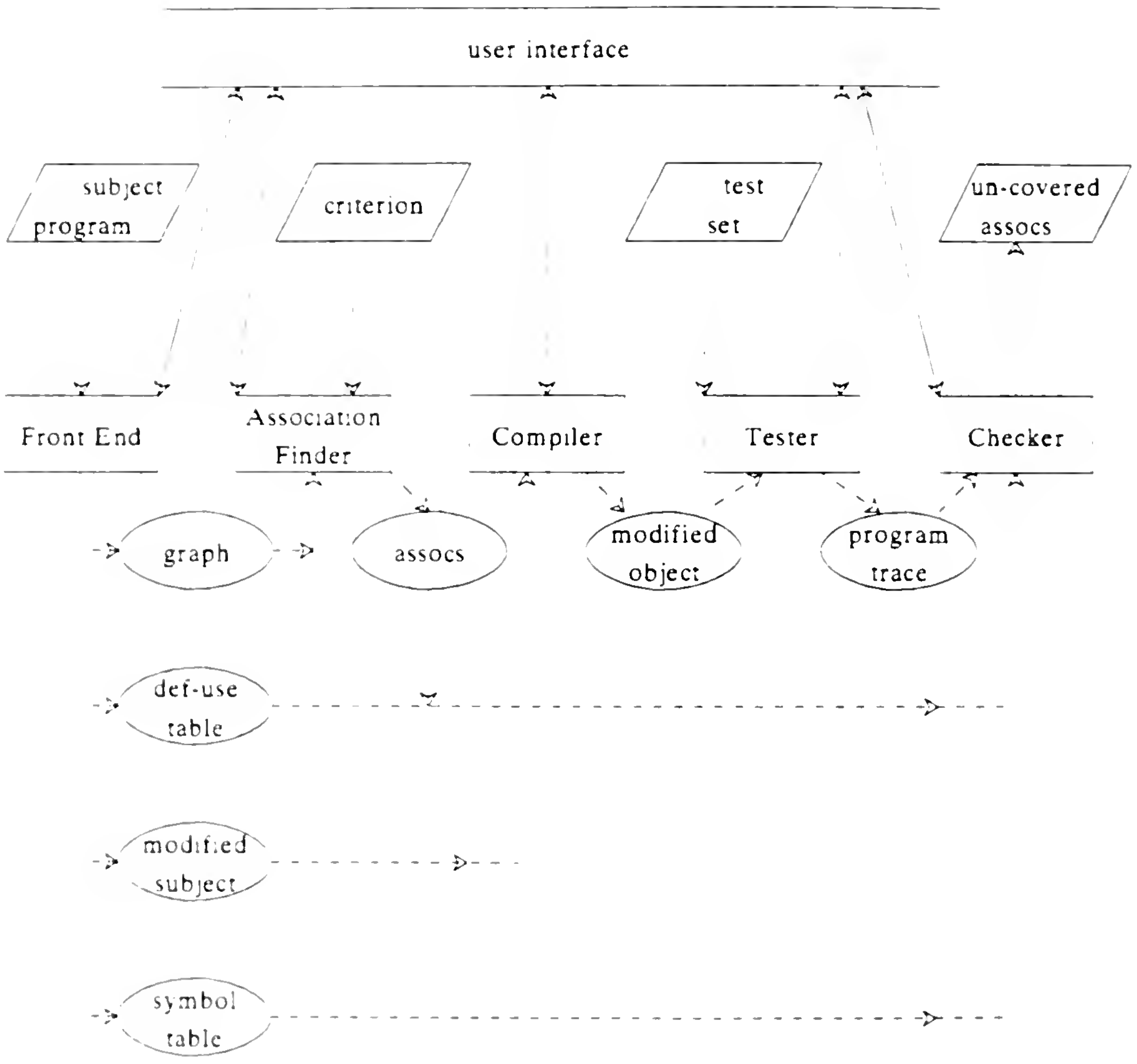


FIGURE 3.2.1

The Design of ASSET

Q is divided into basic blocks and the flow graph is built according to the rules in Figure 2.1.1. Each node in the flow graph, or equivalently each basic block (including certain empty basic blocks), is given a unique number.

The statements in each basic block b of Q are examined. The sequence of definitions and c-uses of variables occurring in the b are recorded and the p-uses of variables on edges whose source is b are recorded according to the rules in Figure 2.1.1. It is not necessary to completely parse the expressions occurring in Q, only to identify the variables occurring in each expression.

The modified subject program is obtained from P by adding a *probe* at the beginning of each basic block in Q, which when executed, will write the block's number to a file called the *program trace*. A probe is which appends a zero to the program trace each time a call to Q is about to terminate is also added to the modified subject program. Extra **begins** and **ends** are added as needed. In order to place a probe in every block, repetitive statements are translated into a form which involves **gotos** and labels. For example, the statement

while x > 0 **do** writeln(x):

in the subject procedure would be translated into a fragment of the form

```
L1:  probe(n1);
     if not (x > 0) then goto L2;
     probe(n2);
     writeln(x);
     goto L1;
L2:  probe(n3):
     probe(n4);
```

where L_1 and L_2 are statement labels, the n_i are node numbers, and the probes write the node numbers to the program trace file. The modified subject program must also include declarations of the program trace file, of labels, of temporary variables introduced while translating **for** loops, and of any other identifiers which may appear in the probes.

Module B: Association-Finder

This module takes as input the def-use table and flow graph produced by the front end, and one of the data flow testing criteria and produces a list of all of the definition-use associations (of the appropriate kind) in subprogram Q. If the criterion is all-defs, all-p-uses, all-c-uses, all-p-uses/some-c-uses, all-c-uses/some-p-uses, or all-uses then the relevant associations are definition-c-use and/or definition-p-use associations. If the criterion is all-du-paths, the relevant associations are du-paths. Since the number of du-paths may be exponential in the number of nodes in the flow graph, ASSET does not prepare a list of them unless necessary.

There are several possible ways to build the list of definition-c-use and definition-p-use associations. The problem can be viewed as finding all of the definitions which reach each use of a variable (use-definition chaining) or as finding all of the uses which can be reached by each definition of a variable (definition-use chaining). There are several well-known polynomial-time algorithms for these problems [AHO86,HEC77].

Finding all of the du-paths in Q has an exponential lower bound. This is because the number of du-paths is potentially exponential, and each of the du-paths must be traversed explicitly.

Module C: Compiler

This module is an ordinary Pascal compiler. In ASSET, it takes the modified subject program as input and produces an object file which we will call the *modified object code*.

Module D: Tester

This module takes as input the modified object code, and a set T of test data. For each test case t in T, it executes the modified program with t as an input. This produces the output P(t) which would be produced by running P on t, and also appends the path (or paths) executed through Q to the program trace file.

Module E: Checker

This module checks which of the associations required by the criterion have been covered by the test set T. It then determines whether the criterion has been satisfied and, if not, outputs a list of associations which required by the criterion but which have not been covered.

We associate with each definition-c-use association and each definition-p-use association a regular expression which describes the set of paths covering the association. Let A be the alphabet {1,...,N} where Q's flow graph has N nodes. For each variable v, let $D^v \subseteq A$ be the set of node numbers corresponding to nodes which have a global definition of v. Let (d,u,v) be a def-c-use association. The regular expression $r^{d,u,v}$=A*d(A−D^v)*uA* describes the set of paths which cover the association in the sense that if π is a path from the entry to the exit of Q then π covers (d,u,v) if and only if π belongs to the language generated by $r^{d,u,v}$. Similarly, the regular expression $r^{d,s,t,v}$=A*d(A−D^v)*stA* describes the set of paths which cover the def-p-use association (d,(s,t),v).

If the criterion is all-defs, all-c-uses, all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, or all-uses, ASSET simulates the operation of deterministic finite automata which correspond to these regular expressions. By submitting each of the paths traversed by the test data as input to each of the automata, ASSET determines which of the def-c-use and which of the def-p-use associations have been covered by the test set. The algorithm in Figure 3.2.2 simulates the parallel operation of the automata. Note that only one pass over the program trace file is needed to determine which of the def-c-use and def-p-use associations are covered by the test set.

If the criterion is all-c-uses, all-p-uses, or all-uses, ASSET then simply checks whether all of the automata corresponding to the appropriate type of use have accepted at least one input. Those which have not accepted correspond to associations which have not been covered. If the criterion is all-defs, all-c-uses/some-p-uses, or all-p-uses/some-c-uses, then a little more bookkeeping is necessary. For example, for the all-defs criterion, it is necessary to check whether *any* of the associations arising from a particular definition of a variable have been covered.

If the criterion is all-du-paths, ASSET uses pattern-matching to determine which du-paths have been covered by the test set. For each du-path δ and each path π in the program trace, ASSET treats π as text, and δ as a pattern, and searches the text to determine whether the pattern occurs. The pattern δ occurs in π if and only if the path π covers the du-path δ. By using a pattern matching algorithm which never has to back-up in the text (such as the Knuth-Morris-Pratt Algorithm [KNU77]), and by checking the patterns represented by all of the du-paths in parallel (in a manner similar to the above simulation of the parallel execution of several automata) ASSET can check all of the du-paths in a single pass over the program trace file.

Limitations

One limitation of this design is that the program traces do not distinguish between

```
const   V =       number of variables;
        N =       number of nodes in flow graph;
        CU =      number of def-c-use associations;
        PU =      number of def-p-use associations;
type    VariableRange:  1..V;
        NodeRange:      1..N;
        C_UseRange:     1..CU;
        P_UseRange:     1..PU;
        StateType:      (q1,q2,q3);
        automaton1 =    record
                                v:              VariableRange;
                                d,u:            NodeRange;
                                state:          StateType;
                                accepted:       boolean;
                        end;
        automaton2 =    record
                                v:              VariableRange;
                                d,s,t:          NodeRange;
                                state:          StateType;
                                accepted:       boolean;
                        end;

var     def_c_use:      array[C_UseRange] of automaton1;
        def_p_use:      array[P_UseRange] of automaton2;
        i:              integer;
begin
        for i:=1 to CU do
                initialize def_c_use[i] so that the fields d,u, and v
                represent the node with the definition, node with the c-use,
                and variable, respectively in the i^th def-c-use association
                and initialize state to q1 and accepted to false;
        for i:=1 to PU do
                initialize def_p_use[i] so that the fields d,s,t, and v
                represent the node with the definition, source and target of the edge with the p-use,
                and variable, respectively in the i^th def-p-use association
                and initialize state to q1 and accepted to false;
        read(program_trace,node);
        while node <> END_OF_PATH_MARKER do
        begin {process next node in the path}
                for i:=1 to CU do {update state of i^th "def-c-use" automaton}
                with def_c_use[i] do
                        if not accepted then
                        case state of
                        q1: if node = d then state := q2;
                        q2: if node = u then accepted := true
                           else if ((node∈D^v) and (node <> d)
                                  then state := q1;
                        end; {case}
                end; {for}
                for i:=1 to PU do {update state of i^th "def-p-use" automaton}
                with def_p_use[i] do
                        if not accepted then
```

```
                    if (d <> s) then
                    case state of
                    q1: if node = d then state := q2;
                    q2: if node = s then state := q3
                       else if ((node∈$D^v$) and (node <> d))
                               then state := q1;
                    q3: if node = t then accepted := true
                       else if node = s then state := q3
                       else if ((node∈$D^v$) and (node <> d))
                               then state := q1
                       else state := q2;
                    end {case}
                    else {d = s}
                    case state of
                    q1: if node = d then state := q3;
                    q3: if node = t then accepted := true
                       else if (node <> d) then state := q1
                    end; {case}
            end; {for}
        read(program_trace,node);
        end; {while}
end;
```

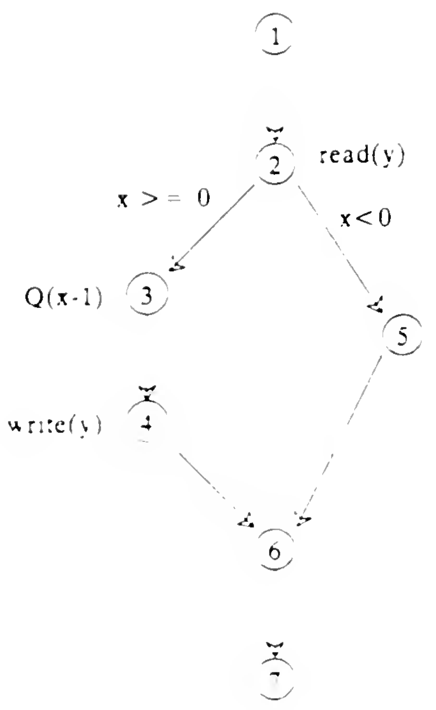
Figure 3.2.2

different invocations of the same procedure. This presents certain problems when the procedure being tested calls itself recursively. For example, consider the following program:

```
program driver(input,output);
var a : integer;
procedure Q(x : integer);
    var y : integer;
    begin
        read(y);
        if x >= 0 then
        begin
            Q(x-1);
            write(y)
        end
    end; {Q}
begin {driver}
    read(a);
    Q(a)
end.
```

The flow graph for procedure Q is shown in Figure 3.2.3. The test set T={(0)} executes the path (1,2,3,4,6,7) through its first invocation of procedure Q, and the path (1,2,5,6,7) through its second invocation of procedure Q. The first of these has a definition-clear subpath with respect to y from node 2 to node 4, hence covers the def-c-use association (2,4,y). However, the design of ASSET described here would say that the paths executed were (1,2,3,1,2,5,6,7) and (4,6,7). Therefore it would not see that the association (2,4,y) had been covered. It is left to the user to determine by hand that the association has been covered.

A more sophisticated design could circumvent this problem by *dynamically* tagging the node numbers in the program trace with a procedure-invocation-ID, then separating the portions of the program trace which arise from different invocations. In the above example, the initial program trace would be (1.1,1.2,1.3,2.1,2.2,2.5,2.6,2.7,2.0,1.4,1.6,1.7,1.0), where each element of the sequence



Flow graph for procedure Q

FIGURE 3.2.3

has the form *invocation-ID.node-number*, and where node number 0 indicates termination of the corresponding invocation. This trace could then easily be divided into the paths (1,2,3,4,6,7) and (1,2,5,6,7), as desired. It is not clear whether the benefits of such a design justify the considerable overhead involved.

3. The Implementation of ASSET

In this section we describe the implementation of the current version of ASSET, which is an experimental prototype.

ASSET runs BSD 4.2 UNIX systems and has been implemented on DEC VAX's and on Sun Workstations*. The front end, the association-finder, and the checker are written in Berkeley Pascal. Each of these is an individual Pascal program. The user interface is written in C and makes system calls to invoke the modules, to manipulate files, and to invoke various additional functions which were not described in the above design description. Module C, the compiler, is the Unix system's Pascal compiler, *pc*. Module D, the checker is implemented as part of the user interface.

The user interface is user-friendly and is driven by commands issued by the user. In a typical ASSET session, the user will initially select a criterion and direct ASSET to invoke the front end, the association-finder, and the compiler. At this point the checker can be invoked to produce a list of all of the definition-use associations. The user then invokes module D, the tester, to run the modified subject program on one or more test cases, then the checker to determine which (if any) associations required by the criterion have not been executed by the test data. At this point, the user can add additional test

* UNIX is a trademark of AT&T Bell Laboratories. DEC and VAX are trademarks of Digital Equipment Corporation. Sun Workstation is a trademark of Sun Microsystems, Inc.

cases to the test set and run the program again or can select a new criterion and check the current test set against the new criterion. A more complete description of ASSET from the user's viewpoint can be found in the ASSET User Manual, included in this thesis as Appendix I.

Module A, the front end, is implemented as a one-pass recursive-descent parser which builds the symbol table, builds the flow graph for Q; classifies variable occurrences as being definitions, c-uses, or p-uses, storing this information in a table called the *def-use table*; and creates a preliminary version of the modified subject program in which the probes have been inserted. Declarations are added in one pass over the modified subject program. The def-use table, an adjacency list representation of the flow graph, and relevant parts of the symbol table are written to files. The front end does not completely parse expressions, nor does it explicitly build the parse tree.

In Module B, the association-finder, a series of depth-first-searches is performed (unless the criterion is all-du-paths.) For each definition of a variable v having a definition in node i, a depth first search is performed beginning at node i. Each time a node j is visited, ASSET checks the def-use table to see whether variable v occurs in node j. If node j has a c-use of v then the def-c-use association (i,j,v) is recorded. If node j is the source of edges $(j,k_1),\ldots,(j,k_m)$ having p-uses of v then the def-p-use associations $(i,(j,k_1),v),\ldots,(i,(j,k_m),v)$ are recorded. If node j has a definition of v then the search backtracks.

At most NV depth first searches are performed where N is the number of nodes in the flow graph and V is the number of variables. Since the graphs involved are sparse, each depth first search takes O(N) time. Thus the complexity of the association-finder is $O(N^2V)$.

This implementation of the association-finder is a hold-over from an earlier version of ASSET (which took as input programs written in a simple language having only simple variables) in which some data flow anomaly detection was performed. More efficiency could be achieved in practice by using one of the algorithms mentioned in section 2.

If the criterion is all-defs, all-c-uses, all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, or all-uses, then module E, the checker, uses the algorithm in Figure 3.2.2. If the criterion is all-du-paths then the checker uses the Knuth-Morris-Pratt pattern matching algorithm. The du-paths are stored internally as an array of arrays of characters and one pass is made over the program trace file.

We conclude this section by mentioning some of the limitations of the current implementation of ASSET. The fact that ASSET stores various tables internally as arrays imposes certain restrictions on the number of nodes in the flow graph, the number of def-use associations, etc. ASSET also places certain minor restrictions on identifiers which can be used in the subject program, in order to avoid interference with the names of files created by ASSET and with identifiers added to the modified subject program. Details of these restrictions can be found in the ASSET User Manual. In addition, the User Manual documents certain bugs in the current implementation which will be corrected in the next version.

CHAPTER 4:

The Feasible Data Flow Testing Criteria

1. Definitions of the feasible data flow testing criteria

Given a subprogram P and a DF criterion C it may be the case that no test data for P exists which satisfies C. This occurs when none of the paths which cover a particular association required by C are executable. In such a case, P cannot be adequately tested according to C. In this section we introduce a new family of criteria which are derived from the DF criteria and which circumvent this problem and investigate some of its properties. We assume that all side effects and aliasing are known.

Recall that a *complete path* is a path from the entry node to the exit node of a subprogram's flow graph. We will say that a complete path is *executable* or *feasible* if there exists some assignment of values to input variables, non-local variables, and parameters which causes the path to be executed. We will say that a path is executable if it is a subpath of an executable complete path. According to this definition, the question of whether or not a given path through a subprogram is executable is independent of the context in which that subprogram is called. However, it may depend on the effects of any procedures or functions which are called along the path. In section 5 we will discuss the consequences of modifying this notion of executability to take into account information about the context in which the subprogram is called. Note that whether or not a particular path is executable depends on the actual subprogram, not just on its def-use graph.

We will say that an association is *executable* if there is some executable complete path which covers it, otherwise it is *unexecutable*. We define subsets fdcu(x,i) $\subseteq$ dcu(x,i) and fdpu(x,i) $\subseteq$ dpu(x,i) whose elements correspond to those associations which are executable as follows: *dcu(x,i)*={nodes j such that x$\in$c-use(j) and there is an *executable* definition clear path with respect to x from i to j}. *dpu(x,i)*={edges (j,k) such that x$\in$p-use(j,k) and there is an *executable* definition clear path with respect to x from i to (j,k)}. Equivalently, fdcu(x,i)={j$\in$dcu(x,i)|the association (i,j,x) is executable} and fdpu(x,i)={(j,k)$\in$dpu(x,i)|the association (i,(j,k),x) is executable}. For each DF criterion C we define a new criterion C* by selecting the required associations from fdcu(x,i) and fdpu(x,i) instead of from dcu(x,i) and dpu(x,i). Precise definitions of these criteria are given in Figure 4.1.1. We will refer to the criteria {(all-du-paths)*, (all-uses)*, (all-p-uses/some-c-uses)*, (all-c-uses/some-p-uses)*, (all-p-uses)*} as *feasible data flow testing criteria*, or FDF criteria, for short.

The FDF criteria satisfy the applicability property: For any subprogram P and any FDF criterion C*, there is some test set T which satisfies C*. However, the question of whether a particular T satisfies C* for subprogram P is undecidable. In going from the family DF to the family FDF, we have traded the undecidability of the existence question, "is there any test which is C-adequate for P?" for the undecidability of the recognition problem "is a given test C*-adequate for P?"

Observe that for any DF criterion C, C $\Rightarrow$ C*. We now investigate the inclusion relations among the FDF criteria.

THEOREM 4.1.1: The family of FDF criteria is partially ordered by strict inclusion as shown in Figure 4.1.2. Furthermore, FDF criterion C_i* includes FDF criterion C_j* if and only if it is explicitly shown to do so in the figure or it follows from the transitivity of the relations.

THE FEASIBLE DATA FLOW TESTING CRITERIA

fdcu(x,i) = {j∈ dcu(x,i)| the association (i,j,x) is executable}
fdpu(x,i) = {(j,k)∈ dpu(x,i)| the association (i,(j,k),x) is executable}

A test-set/driver-program pair (T,D) satisfies criterion C for subprogram P if and only if for each node i in P's flow graph and each x ∈ def(i) the set Π of paths executed by T covers the following associations:

CRITERION	REQUIRED ASSOCIATIONS
(all-defs)*	if fdcu(x,i) ∪ fdpu(x,i) ≠ ϕ then some (i,j,x) s.t j∈fdcu(x,i) or some (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i).
(all-c-uses)*	all (i,j,x) s.t. j∈ fdcu(x,i).
(all-p-uses)*	all (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i).
(all-p-uses/some-c-uses)*	all (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i). In addition, if fdpu(x,i) = ϕ and fdcu(x,i) ≠ ϕ then some (i,j,x) s.t. j∈ fdcu(x,i).
(all-c-uses/some-p-uses)*	all (i,j,x) s.t. j∈ fdcu(x,i). In addition, if fdcu(x,i) = ϕ and fdpu(x,i) ≠ ϕ then some (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i).
(all-uses)*	all (i,j,x) s.t. j ∈ fdcu(x,i) and all (i,(j,k),x) s.t. (j,k) ∈ fdpu(x,i).
(all-du-paths)*	all executable du-paths with respect to x from i to j s.t. j∈ dcu(x,i) and all executable du-paths with respect to x from i to (j,k) for each (j,k) ∈ dpu(x,i).

For comparison we also define the criteria (all-nodes)* [(all-edges)*, (all-paths)*, respectively] which require that Π cover each executable node [each executable edge, each executable path, respectively.]

Figure 4.1.1

$$\begin{tikzcd}[column sep=small]
 & & \text{(ALL-PATHS)}^* \arrow[dl, Rightarrow] \arrow[dd, Rightarrow] \arrow[dr, Rightarrow] & & \\
 & \text{(ALL-DU-PATHS)}^* & & \text{(ALL-EDGES)}^* \arrow[d, Rightarrow] & \\
 & & \text{(ALL-USES)}^* \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \text{(ALL-NODES)}^* & \\
 & \text{(ALL-C-USES/SOME-P-USES)}^* \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & & \text{(ALL-P-USES/SOME-C-USES)}^* \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\text{(ALL-C-USES)}^* & & \text{(ALL-DEFS)}^* & & \text{(ALL-P-USES)}^*
\end{tikzcd}$$

FIGURE 4.1.2

The relationship among the feasible data flow testing criteria

PROOF:

A. Strictness of the inclusions

We first observe that if subprogram P has no unexecutable paths, then a test is C-adequate for P if and only if it is C*-adequate for P. This observation, along with the proofs of strictness of the inclusions in Theorem 1 of [RAP85], none of which involve subprograms with unexecutable paths, shows that all of the inclusions in Figure 4.1.2 are strict. It thus suffices to show that the inclusions in Figure 4.1.2 hold.

B.1. (all-paths)* $\Rightarrow$ (all-uses)* :

Suppose not. Then there is a subprogram P and a set T of test data which is (all-paths)*-adequate for P but not (all-uses)*-adequate. Let Π be the set of paths through P which T executes. There exist a node i in P with a global definition of some variable x, a node j with a global c-use of x or edge (j,k) with p-use of x, and an *executable* definition clear path with respect to x from i to j [respectively from i to (j,k)] which is not covered by Π. This contradicts the fact that Π covers every executable path.

The proofs that (all-paths)* $\Rightarrow$ (all-du-paths)* and (all-paths)* $\Rightarrow$ (all-edges)* are similar.

B.2. (all-paths)* $\Rightarrow$ (all-du-paths)* :

Suppose not. Then there is a subprogram P and a set T of test data which is (all-paths)*-adequate for P but not (all-du-paths)*-adequate. Let Π be the set of paths through P which T executes. There exists an *executable* du-path which is not covered by Π. This contradicts the fact that Π covers every executable path.

B.3. (all-paths)* $\Rightarrow$ (all-edges)* :

Suppose not. Then there is a subprogram P and a set T of test data which is (all-paths)*-adequate for P but not (all-edges)*-adequate. Let Π be the set of paths through P which T executes. There exists an executable edge (i,j) which is not covered by Π. This contradicts the fact that Π covers every executable path.

B.4. (all-edges)* $\Rightarrow$ (all-nodes)* :

Let T be a test which satisfies (all-edges)* for subprogram P, and let Π be the set of paths executed by T. Let n be any executable node in P. If n is the entry node, then n has a unique successor, m, and (n,m) is executable. So Π covers (n,m) and hence covers n. If n is not the entry node, then since n is executable, some branch (i,n) is executable. So Π covers (i,n) and hence covers n.

B.5. (all-uses)* $\Rightarrow$ (all-p-uses/some-c-uses)*, (all-p-uses/some-c-uses)* $\Rightarrow$ (all-p-uses)*, (all-p-uses/some-c-uses)* $\Rightarrow$ (all-defs)*, (all-uses)* $\Rightarrow$ (all-c-uses/some-p-uses)*, (all-c-uses/some-p-uses)* $\Rightarrow$ (all-defs)* :

These inclusions follow immediately from the definitions of the criteria given in Figure 4.1.1. For example, any set Π of paths which covers all of the associations required by (all-uses)* will *a fortiori* cover all of the associations required by (all-p-uses/some-c-uses)*.

We next show that those relations not in the transitive closure of the diagram in Figure 4.1.2 do not hold.

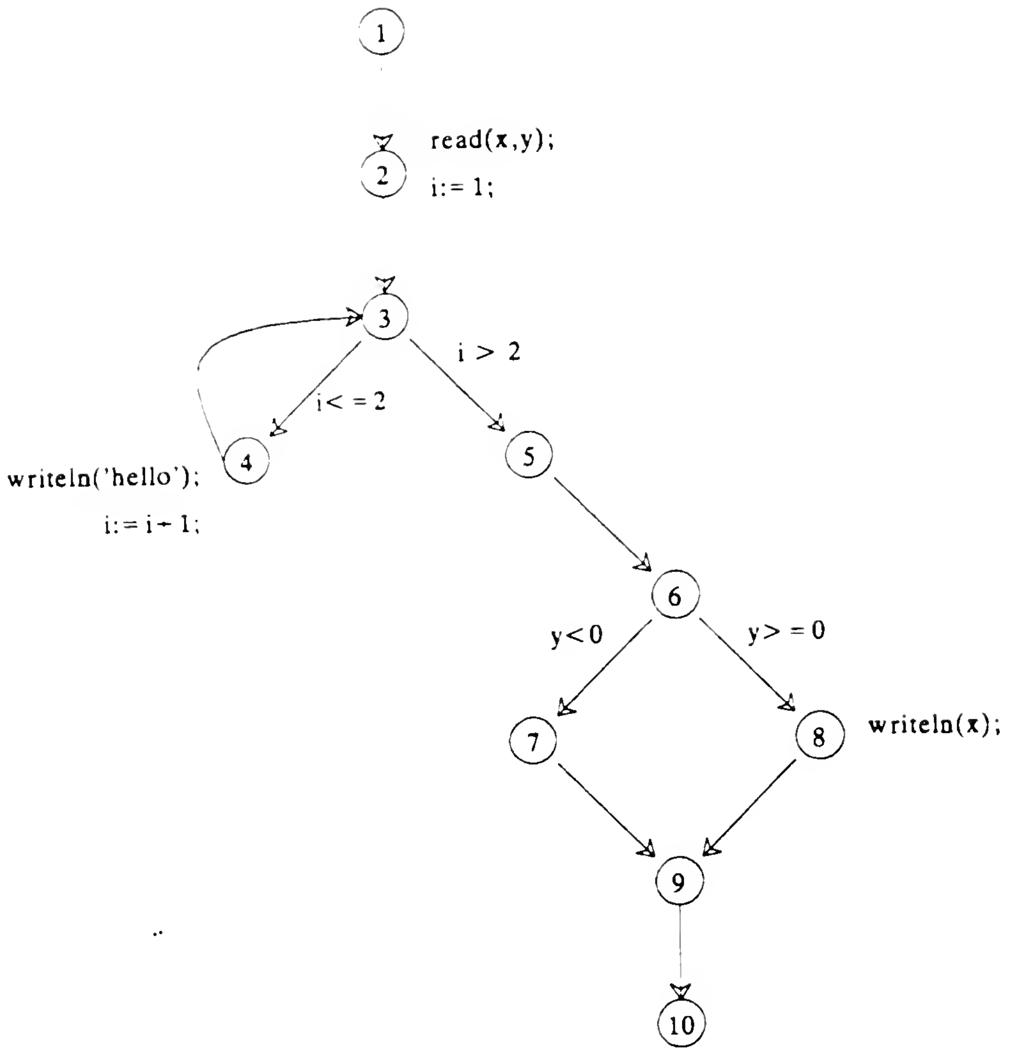
C.1. (all-du-paths)* (all-p-uses)*; (all-du-paths)* (all-p-uses/some-c-uses)*; (all-du-paths)* (all-uses)*; (all-du-paths)* (all-c-uses/some-p-uses)*; (all-du-paths)* (all-defs)*; (all-du-paths)* (all-edges)*; (all-du-paths)* (all-nodes)* :

It suffices to show that (all-du-paths)* (all-p-uses)*, (all-du-paths)* (all-defs)*, and (all-du-paths)* (all-nodes)*. The rest follows from the transitivity of $\Rightarrow$. Consider the subprogram shown in Figure 4.1.3a. Its du-paths are shown in Figure 4.1.3b. Of these, only (1,2), (2,3,4), (4,3,4), and (4,3,5) are executable. Let $T = \{(X,Y)\}$ where X is any integer and $Y < 0$. Since T executes $\Pi = \{(1,2,3,4,3,4,3,5,6,7,9,10)\}$, T satisfies (all-du-paths)*. However, Π does not cover the associations (2,(6,8),y), (2,8,x), or node 8, all of which are covered by the executable path (1,2,3,4,3,4,3,5,6,8,9,10), so T does not satisfy (all-p-uses)*, (all-defs)*, or (all-nodes)*.

Intuitively, (all-du-paths)* fails to include these criteria because it is possible for a subprogram to have certain definition-use associations which can be executed only by paths which traverse some loop one or more times.

C.2. (all-p-uses)* (all-edges)*; (all-p-uses/some-c-uses)* (all-edges)*; (all-uses)* (all-edges)*; (all-p-uses)* (all-nodes)*; (all-p-uses/some-c-uses)* (all-nodes)*; (all-uses)* (all-nodes)* :

Consider the subprogram in Figure 4.1.4, where *y* is a local variable (and hence *does not* have a definition in the entry node). Notice that since node 3 is unexecutable, y is always uninitialized at node 5. In the absence of any information about which (if either) edge leaving node 5 will be executed when the program is run on actual test data, we make the worst-case assumption that edges (5,6) and (5,7) are both executable. This would be the case, for example, in an environment in which uninitialized variables receive arbitrary values. Since node 3 is unexecutable, the only executable definition-use associations are (1,2,input), (2, (2,4), x) and (2,9,input$\uparrow$). Let T be a test which executes $\Pi=\{(1,2,4,5,6,8,9)\}$ or $\Pi=\{(1,2,4,5,7,8,9)\}$. Then T satisfies (all-p-uses)*, (all-p-uses/some-c-uses)*, and (all-uses)*, but does not satisfy (all-edges)* or (all-nodes)*.



1
read(x,y);
2
i:= 1;
3
i > 2
i< = 2
4
writeln('hello');
i:= i + 1;
5
6
y< 0
y> = 0
7
8
writeln(x);
9
10

FIGURE 4.1.3a

Path	With respect to	executable
(1,2)	input↑	yes
(2,3,4)	i	yes
(2,3,5)	i	no
(4,3,4)	i	yes
(4,3,5)	i	yes
(2,3,5,6,7,9,10)	input↑	no
(2,3,5,6,8,9,10)	input↑	no

Figure 4.1.3b

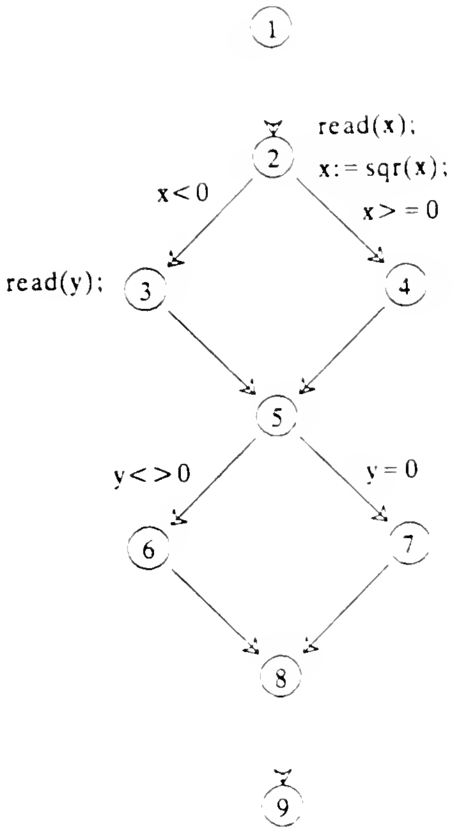


FIGURE 4.1.4

Notice that the subprogram in Figure 4.1.4 has a data flow anomaly [OST76]. We shall see below that this is not a mere coincidence, but that rather, it is this particular kind of anomaly which prevents the inclusions from holding.

The rest of the non-inclusions follow from the incomparability and strictness proofs in [RAP85]. ■

It seems discouraging that (all-p-uses)* fails to include (all-edges)*. Data flow testing was developed in order to "bridge the gap" between branch testing and path testing. Since many "real-life" subprograms cannot be adequately tested using the unstarred versions of the data flow criteria, one would hope that the FDF criteria would "bridge the gap" between (all-edges)* and (all-paths)*. We have seen that this is not the case. We next show that for a certain class of "well-behaved" subprograms, any test which satisfies (all-p-uses)* satisfies (all-edges)*.

DEFINITION: We will say that a subprogram P satisfies the No-Feasible-Undefined-P-uses property (NFUP) if and only if for every executable edge (i,j) in P having a p-use of a variable x there is some *executable* path from the start node to edge (i,j) which contains a global definition of x.

We note that it is quite reasonable to expect subprograms to have property NFUP. If (i,j) is an edge which causes NFUP to fail, then any input which causes (i,j) to be executed will involve referencing an uninitialized variable.

THEOREM 4.1.2 For the class of subprograms which satisfy NFUP, (all-p-uses)* $\Rightarrow$ (all-edges)*.

PROOF: Let P be a subprogram satisfying NFUP, let T be a test which satisfies (all-p-uses)* for P, let Π be the set of paths executed by T, and let (i,j) be an executable edge in P. Suppose (i,j) has a p-use of a variable x. By hypothesis there is an executable path π

from the start node to (i,j) which includes a global definition of x. Let n be the last node in π having a global definition of x. Then (n, (i,j), x) is an executable definition-p-use association so it is covered by Π. Hence (i,j) is covered by Π.

If (i,j) has no p-uses, then the result follows by a straight-forward modification of the corresponding part of the proof of (all-p-uses) $\Rightarrow$ (all-edges) in Chapter 2, section 1.■

In [RAP85] the class of subprograms to which data-flow testing applies is restricted to those subprograms satisfying the No-Syntactic-Undefined-P-use Property, defined in section 2 above. This restriction was necessary in order to insure that all-p-uses $\Rightarrow$ all-edges. NFUP is a strengthening of NSUP. It takes into account the fact that in subprograms satisfying NSUP, the only paths π from the entry node to some p-use of x such that x has a global definition in some node in π might be unexecutable.

It is tempting to restrict the class of programs to which the FDF criteria apply to those satisfying NFUP. It is our feeling however that while one can live with the undecidability of the adequate test *recognition* problem and perhaps (albeit very uncomfortably) with the undecidability of the adequate test *existence* problem, one should at least be able to decide algorithmically whether a given testing strategy *applies* to a given subprogram. Since it is undecidable whether a given subprogram satisfies NFUP, we refrain from requiring this property of subprograms to be tested.

Another possible way to force (all-p-uses)* to include (all-edges)* would be to require subprograms to satisfy the No-Anomalies Property (NA):

Every path from the start node to a use of a variable x must contain a definition of x.

Osterweil and Fosdick [OST76] consider any subprogram not satisfying this property to have data-flow anomaly indicative of possible subprogram error. Since NA is a purely syntactic property and NA implies NFUP we could restrict FDF testing to subprograms satisfying this property. We feel that this is overly restrictive, since many perfectly good subprograms fail to satisfy NA.

One way to force NA to be satisfied is to give the entry node a definition of each variable. This would potentially increase the number of def-use associations and thus make the criteria more demanding. However, it would make the model of the subprogram's data flow reflect the actual data flow less accurately.

Another approach is to perform FDF testing in conjunction with a check for data-flow anomalies. For any subprogram which satisfies NA and any test set T which satisfies (all-p-uses)* the tester will be assured that T satisfies (all-edges)* . In case NA does not hold, the tester should explicitly check whether (all-edges)* is satisfied and if necessary add more test data or inspect the subprogram for references of uninitialized variables.

2. Axiomatic Evaluation of the Feasible Data Flow Testing Criteria

In this section we evaluate which of the axioms for a good adequacy criterion are satisfied by the feasible data flow testing criteria.

Several of the axioms are of the form "There exists a program P and a test set T such that some property holds". If a data flow testing criterion C satisfies one of these axioms, then the corresponding feasible data flow testing criterion C* automatically satisfies the axiom. In these cases, the examples used in Chapter 2.2 to prove that C satisfies an axiom also show that C* satisfies the axiom.

Axiom 1: Applicability

For every program, there exists an adequate test set.

The feasible data flow testing criteria have all been defined in such a way as to guarantee that the applicability criterion is satisfied. For each of these criteria, the definition-use associations which must be covered by the test set are all executable. By definition, this guarantees the existence of adequate test data.

Axiom 2: Non-Exhaustive Applicability

There is a program P and test set T such that P is adequately tested by T and T is not an exhaustive test set.

All of the feasible data flow testing criteria satisfy this axiom. This follows from the fact that all of the data flow testing criteria satisfy the axiom.

Axiom 3: Monotonicity

If T is adequate for P and $T \subseteq T'$ then T' is adequate for P.

Since all of the executable associations covered by T are also covered by T', all of the data flow criteria satisfy the monotonicity property.

Axiom 4: Inadequate Empty Set

The empty set is not adequate for any program.

Every subprogram has at least one *executable* def-p-use association. To see this, let n be the unique node such that n has at least two successors and no node on any path from the entry node to node n has more than one successor. At least one of the edges (n,m) whose source is node n must be executable. Since node n has more than one successor, edge (n,m) has a p-use of some variable, v. Since the subprogram is assumed to satisfy the No-Syntactic-Undefined-p-uses property, there is a global definition of v on the path from the entry node to (n,m). Therefore, there is an executable definition-p-use association. Note that the path from the entry node to edge (n,m) is a du-path.

It follows that the criteria (all-defs)*, (all-p-uses)*, (all-p-uses/some-c-uses)*, (all-c-uses/some-p-uses)*, (all-uses)*, and (all-du-paths)* satisfy the inadequate empty set property.

The criterion (all-c-uses)* fails to satisfy the inadequate empty set axiom. This is because it is possible for a program to have no *executable* definition-c-use associations. Consider, for example, the program

```
program example(input,output);
var x : integer;
begin
        read(x);
        while x=x do writeln('hello')
end.
```

The only definition-c-use association in this program arises from the fact that there is a definition of *input*↑ in the entry node and a c-use of *input*↑ in the exit node. But this association is unexecutable because control can never reach the exit node.

Axiom 5: Anti-extensionality

There are programs P and Q such that P is equivalent to Q, T is adequate for P, but T is not adequate for Q.

All of the feasible data flow testing criteria satisfy the Anti-extensionality axiom.. This follows from the fact that all of the DF criteria satisfy the axiom.

Axiom 6: General Multiple Change

There are programs P and Q which are the "same shape", and a test set T such that T is adequate for P, but T is not adequate for Q.

All of the feasible data flow testing criteria satisfy the general multiple change axiom. This follows from the fact that all of the DF criteria satisfy the axiom.

Axiom 7: Antidecomposition

There is a subprogram P which calls a subprogram Q and a test T such that T is adequate for the program P' obtained from P by expanding in-line one call to Q, but the test-set/driver-program pair (T,P) is not adequate for Q.

All of the feasible data flow testing criteria satisfy the antidecomposition axiom. Recall that most of the DF criteria failed to satisfy this axiom because (with certain exceptions, which are artifacts our model of data flow) each definition-use association a in Q corresponds to a definition-use association a' in the portion of P' which corresponds to Q. However, it is possible for a to be executable, while a' is unexecutable. An extreme case of this occurs when every call to Q from P is unexecutable.

Consider the following programs, whose flow graphs are shown in Figure 4.2.1.

```
program P(input,output);
var x : integer;
procedure Q;
        var y: integer;
        begin {Q}
                read(y);
                if y>=0 then writeln('hello')
                else writeln('goodbye')
                writeln(y);
        end; {Q}
begin
        read(x);
        if x >=0 then
```

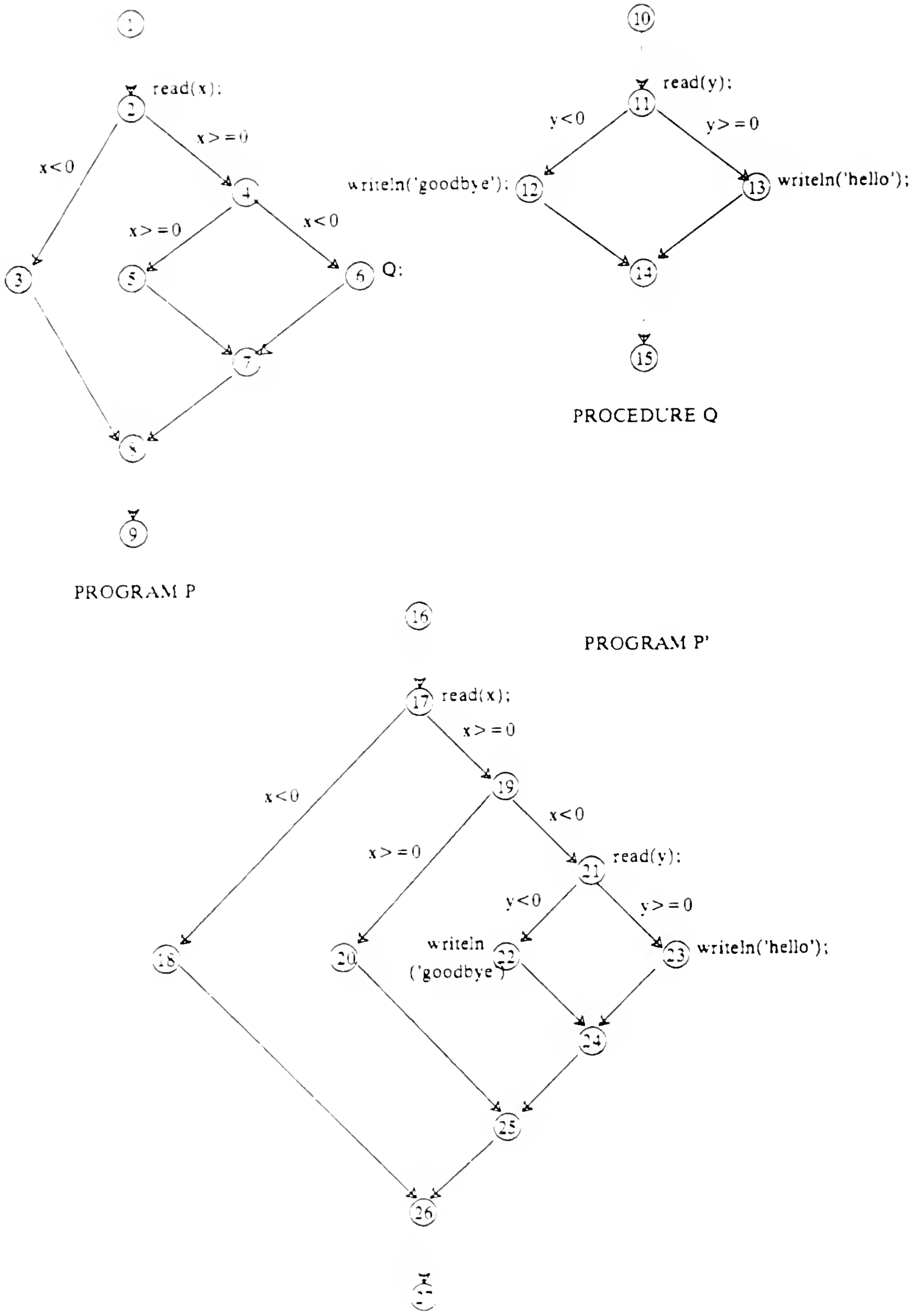
```
          if x < 0 then Q
end.


program P'(input,output);
var x,y : integer;
begin
     read(x);
     if x >=0 then
          if x < 0 then
          begin {procedure Q expanded in-line}
                read(y);
                if y>=0 then writeln('hello')
                else writeln('goodbye')
                writeln(y);
          end; {end of in-line expansion of Q}
end.
```

Note that the call to Q in P is unexecutable. The only executable paths from the entry to the exit of *P′* are (16,17,18,26,27) and (16,17,19,20,25). Since the test set T={(−1),(0)} executes both of these paths, it covers all of the *executable* definition-use associations in *P′*, and hence satisfies all of the FDF criteria. When the elements of T are input to program P, however, they do not cover any of the associations (11,14,y), (11,(11,12),y), or (11,(11,13),y) in Q. Note that these associations are executable, since they can be covered, for example by the driver-program/test-set pair (*P″*,T) where *P″* just calls Q. So T does not satisfy all-defs or all-p-uses for Q, and hence does not satisfy any of the criteria. This shows that all of the FDF criteria satisfy the Antidecomposition axiom.

Recall that the DF criteria all-defs, all-c-uses, and all-c-uses/some-p-uses satisfied this axiom as an only artifact of the particular model of data flow employed. In the case of the FDF criteria, on the other hand, the criteria satisfy the axioms because of the fundamental nature of the criteria.

FIGURE 4.2.1


read(x);
x<0
x>=0
x>=0
x<0
Q:
PROGRAM P
read(y);
y<0
y>=0
writeln('goodbye');
writeln('hello');
PROCEDURE Q
PROGRAM P'
read(x);
x>=0
x<0
x>=0
x<0
read(y);
y<0
y>=0
writeln ('goodbye')
writeln('hello');

Axiom 8: Anticomposition

There is a program R whose statement part consists of a call to procedure P followed immediately by a call to procedure Q, and a test set T for R such that is adequate for P and for Q (as procedure's called by R) but T is not adequate for the program R' obtained from R by expanding P and Q in-line.

All of the feasible data flow testing criteria satisfy this axiom. This follows from the fact that the DF criteria satisfy the axiom.

Combining the results in this section gives us

THEOREM 4.2.1:

The criteria (all-defs)*, (all-p-uses)*, (all-p-uses/some-c-uses)*, (all-c-uses/some-p-uses)*, (all-uses)*, and (all-du-paths)* all satisfy the applicability, non-exhaustive applicability, monotonicity, inadequate empty set, anti-extensionality, general multiple change, anti-decomposition, and anti-composition axioms. The (all-c-uses)* criterion satisfies all of the axioms except for the inadequate empty set axiom.■

3. A generalized notion of executability

The definition of executability given in Section 3 fails to take into account any information about the context in which a subprogram is called. It may be the case that there is no input data to the program as a whole which cause the execution of a particular executable path through a subprogram. In order to adequately test such a subprogram with respect to a given FDF criterion it may be necessary to write a driver program which assigns particular values to global variables and parameters, and then calls the subprogram. Whether this extra effort is "worthwhile" depends on whether it is likely that the subprogram will ever be called in a context other than the one in which it currently appears in the program. In this section we define a more general notion of executability which takes into account information about the context in which a subprogram is called and explore the effects of this generalization on the FDF criteria.

Consider the program:

```
program main(input,output);
type CharString  = array[1..10] of char;
var string1 : CharString;
      length : integer;


procedure WriteString(str: CharString; n : integer);
{Writes the first n characters of str to standard output.}
var i : integer;
begin
   for i := 1 to n do write(str[i])
end;

begin {statement part of main program}


   if length > 0 then WriteString(string1,length)
   else  ...


end. {main }
```

Suppose that at every point in the program at which *WriteString* is called, the value of *n* is guaranteed to be strictly greater than zero. Then no input to the program can cause the execution of the path through the procedure which traverses the loop zero times.

In order to adequately test *WriteString* with respect to the criterion (all-uses)* it is necessary to include test data which causes the **for** loop to be traversed zero times. To do this, one must write a driver program which calls *WriteString* with the second parameter having a value less than or equal to zero. If we think that we might actually want to use the procedure *WriteString* in a less restricted context (for example due to modifications of the calling program or to re-using the procedure in a different program) then this is a reasonable thing to do. On the other hand, if we are fairly certain that the procedure will never be called in a context where *n* is less than or equal to zero, then writing a driver program could be construed as being a wasted effort. What is needed is a notion of test data adequacy which takes into account information about the context in which the subprogram being tested can be called.

We can achieve this by relativizing the definition of executability as follows. We associate with the subprogram to be tested, a predicate $IC(V_1,...,V_k)$, called the *input constraint* where $V_1,...,V_k$ represent the subprogram's parameters and non-local variables. A path through the subprogram is then *executable relative to IC* if there exists some assignment of values to input variables, parameters, and non-local variables which satisfies IC and which causes the path to be executed. A path is executable as defined in Section 4.1 if and only if it is executable relative to the input constraint IC≡TRUE.

The notion of executability of an association and the definitions of the FDF criteria can be relativized in a straight forward manner.

The relationship among the relativized FDF criteria is essentially the same as that among the non-relatived criteria. The definitions must be modified to reflect the fact that the objects being tested now consist of pairs (P,IC) where P is a subprogram and IC is an input constraint. We say that relativized criterion C_1 includes relativized criterion C_2 if for every subprogram/input-constraint pair (P,IC) every test which satisfies C_1 for that pair also satisfies C_2. C_1 strictly includes C_2 if C_1 includes C_2 and for some pair (P,IC) there is a test which satisfies C_2 but does not satisfy C_1. It is easy to show that the relationship among the relativized criteria is as shown in Figure 4.1.2.

One reasonable choice for the input constraint is the predicate IC^{spec} obtained by taking the constraints on the input to the program as a whole (drawn from the program's specification), conjoining them, and "pushing them through" the program to all points at which the subprogram being tested is called. In practice, one might want to use a weaker predicate than IC^{spec} which can be built up during the testing process as follows. Suppose that at some point in the testing process the tester notices that a particular executable association has still not been exercised. Upon examining the program to see what values of input data, non-local variables and parameters would cause the execution of that association, the tester sees that the needed values of non-local variables and parameters cannot arise in the context of the program as a whole. One can then formulate a constraint which reflects this fact and conjoin it to the previous constraint.

If the calling program is modified some time after the subprogram has been certified adequately tested, the predicate IC will provide useful documentation which will help in selecting additional test data for the subprogram. If IC is still satisfied whenever the subprogram is called then no further testing of the subprogram will be needed. If IC no longer hold at the points of call, however, it will be necessary to update IC, determine which def-use associations become executable relative to the new set of constraints, and add test data to exercise those associations.

CHAPTER 5:
The Path Expression Heuristic

1. Introduction

In order to determine whether a test set T satisfies a feasible data flow testing criterion C, it is necessary to examine all of the definition-use associations required by C but not covered by T to see whether any of them are executable. This may be a tedious and error prone task.

In this chapter we present a heuristic called the *path expression method* which attempts to determine whether a given definition-use association (d,u,v) is executable. There are two "hard" aspects to this problem. First, there may be infinitely many paths which cover the association. Second, there is no algorithm to determine whether a particular path is executable. Our approach involves using regular expressions called *path expressions* to describe a set Π of paths which could potentially be executable paths covering the association, and using a combination of symbolic evaluation and data flow information to eliminate from consideration subsets of Π consisting of paths which can be shown to be unexecutable or to have definitions of v.

The approach is heuristic in the sense that it will either answer "association is unexecutable", or "association may be executable". If it answers "association is unexecutable" then the association is guaranteed to in fact be unexecutable. If it answers "association may be executable" then the association may be executable or unexecutable. Hopefully, for "realistic" programs, it will only infrequently answer "association may be executable" when the association is actually unexecutable. When it answers "association may be executable" it will also supply some information which may help the user in determining by hand that the association is unexecutable, or in selecting a test case which covers the association.

Let P be the subprogram being tested. Throughout this chapter, unless otherwise noted, we will make the following assumptions, in addition to the assumptions in Chapter 2, section 1:

1. No aliasing. That is, we assume that when P is called, all of the actual parameters corresponding to **var** formal parameters are distinct from each other and distinct from all global variables which occur in P. We also assume that no pointer variables occur in P. We will relax this assumption later.
2. No hidden side effects. That is, the only variables visible to P which are modified by any procedure or function P' called by P are those variables which are passed to P' as **var** parameters.
3. Functions called by P have no **var** parameters.
4. P has no **repeat-until** statements.

Assumptions 1 and 2 insure that the data flow information is sufficiently accurate. Assumption 3 simplifies the symbolic evaluation which will be performed. Assumption 4 is a technical convenience which insures that the exit from each loop in the flow graph of P is at the top of the loop. In section 5.6 we discuss the effects of relaxing these assumptions.

Loops in the flow graph will play an important role in the path expression method. A node h in a flow graph G is a *loop header*, if, when a depth first search is performed on G, starting at the entry node, h is the target of a back edge. In the recursive definition of structured flow graphs given in Figure 2.1.1, the loop headers are the nodes labelled with the letter "i", in the subgraphs corresponding to **for** statements and **while** statements (and the entry node to S_1 in the subgraph corresponding to the *repeat-until* statement.) If (e,h) is the back edge whose target is h*, then the we define the *loop corresponding to h* to be the subgraph of G consisting of h along with those nodes n such that there is a path from n to e which does not pass through h. A *path through the loop headed by node h* is a path $\pi = (i_1,...,i_k)$, where $k \geq 1$, $i_1 = i_k = h$, and i_j is a node in the loop headed by node h, for $1 \leq j \leq k$. Note that according to this definition, the path $\pi = (h)$ is a path through the loop headed by node h.

We begin this chapter with an overview in Section 5.2 of how the path expression method works. In section 5.3, we define the syntax of path expressions and show how they can be used to represent a set of paths through a flow graph. A new symbolic evaluation technique called *partial symbolic evaluation of path expressions* is described in Section 5.4. A complete description of the method is given in Section 5.5. Some limitations and enhancements are discussed in Section 5.6. An algorithm for the derivation of a path expression representing the set of paths from a given node to a given node or edge is presented in Appendix II. A brief description of our (partial) implementation of the path expression heuristic and a script of a "typical" session are given in Appendix III.

2. Intuition Motivating the Path Expression Method

In this section we introduce the path expression method by presenting two examples.

* Note that for structured graphs, there is only one such edge.

Example 5.2.1:

We begin with a simple example based on a common program fragment:

```
begin
    done := false;
    while not done do
    begin
        S1;
        if B then done := true
        else S2;
        S3;
    end;
end;
```

where S1,S2, and S3 are arbitrary statements and B is a boolean expression. The flow graph for this program fragment is shown in Figure 5.2.1.

Consider the definition-p-use association (1,(2,8),done). Let Π be the set of paths from node 1 to edge (2,8). We can represent this set by a *path expression*

$$\mathbf{1}\cdot\mathbf{p}_2\cdot\mathbf{8}.$$

The symbols **1** and **8** are *node symbols* which represent nodes 1 and 8, respectively. The symbol $\mathbf{p}_2$ is a *loop symbol* which represents the set of paths through the loop headed by node 2. We think of the path expression as meaning "any path in Π consists of node 1 followed by some arbitrary path through the loop whose header is node 2, followed by node 8." This representation of Π is "high-level" in the sense that it omits details about what happens inside of the loop.

For a path π in Π to be executable, the values of the variables must satisfy the expression E≡(*done*=true) when π exits from the loop.

Since the variable *done* is set to *false* in node 1, the expression E cannot be satisfied by the values of the variables at the bottom of node 1. Therefore, along any executable path from 1 to 8, the expression E must change value within the loop.

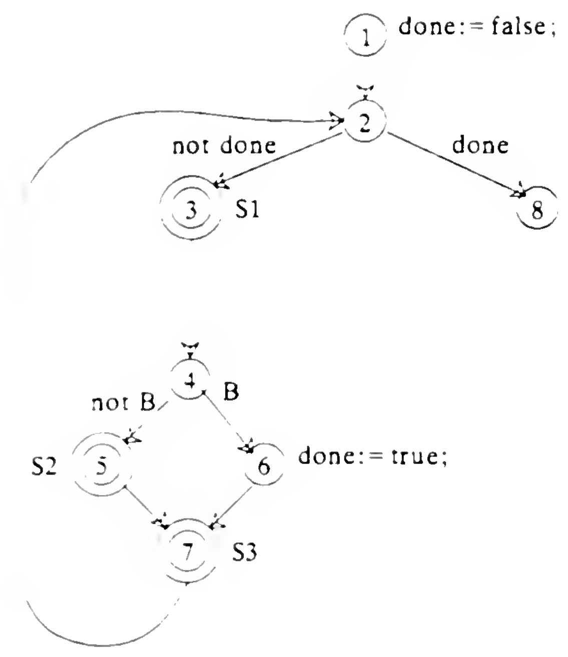


FIGURE 5.2.1

In order for E to change value, one of the variables in E must have a definition. In this case, the only variable in E is *done* so we can eliminate from consideration any path along which *done* has no definition. But we are looking for paths which are definition clear with respect to *done*. so we can also eliminate any paths which do have a definition of *done*. We have eliminated all of the possibilities, so we can conclude that the association is unexecutable.

Note that in this case we did not have to pay attention to any details about what happens within the loop. In other cases we will have to rewrite the path expression to make it express more details about what happens inside of loops. In either case, path expressions provide a convenient formalism which helps us to look at the problem at "the right level of detail".

A similar situation arises from a program fragment consisting of a **for** loop in which the upper bound is always greater than or equal to the lower bound. If the counter variable x is set equal to the lower bound in node i and if edge (j,k) is the exit from the loop, then the def-use association (i,(j,k),x) is unexecutable. An analysis analogous to the one above can be used to prove this. These two common "paradigms" of unexecutable associations can by shown to be unexecutable by a straight-forward application of the path expression method.

Example 5.2.2:

Our next example is somewhat more complicated. Consider the association (2,(6,14),switch) in the "bubble-sort" program shown in Figure 5.2.2. The set of paths from node 2 to edge (6,14) can be represented by the path expression

$$\alpha=2\cdot p_3\cdot 5\cdot p_6\cdot 14.$$

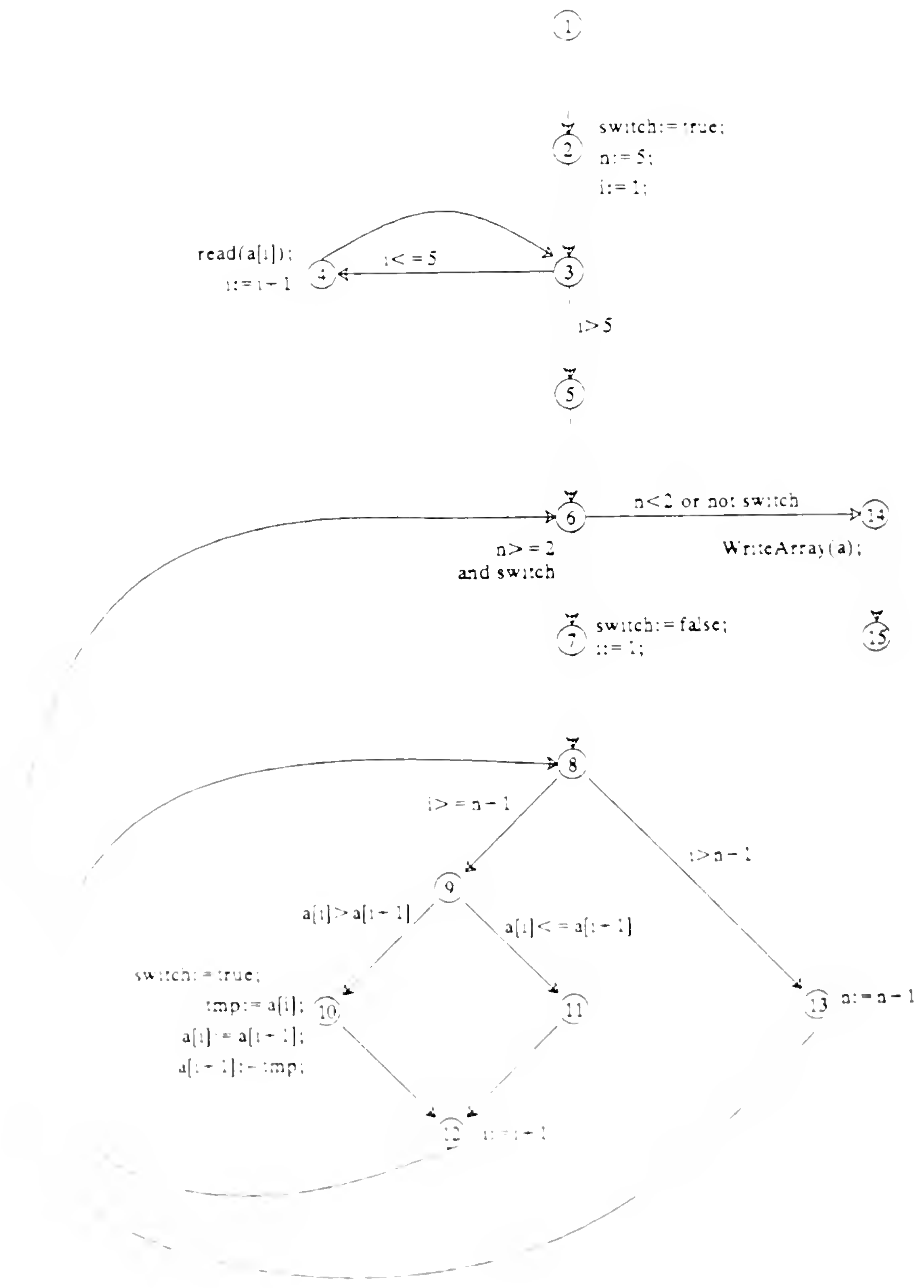

1
2
switch:= true;
n:= 5;
i:= 1;
read(a[i]);
i:= i + 1
4
i< = 5
3
i> 5
5
6
n< 2 or not switch
14
WriteArray(a);
n> = 2
and switch
7
switch:= false;
i:= 1;
15
8
i> = n - 1
i> n - 1
9
a[i] > a[i + 1]
a[i] < = a[i + 1]
switch: = true;
tmp:= a[i];
a[i] := a[i + 1];
a[i + 1]:= tmp;
10
11
13
n:= n - 1
12
i:= i + 1

FIGURE 5.2.2

We think of α as saying that any path from node 2 to edge (6,14) starts at node 2, then takes an arbitrary path through the loop headed by node 3, then goes to node 5, then takes an arbitrary path through the loop headed by node 6, then ends at node 14. In order to execute branch (6,14), the expression

$$E \equiv ((n < 2) \text{ or } (switch = false))$$

must be satisfied by the values of the variables at the bottom of node 6. Since the values of the variables never satisfy E when control reaches the bottom of node 5, we know that on any executable path at least one of the variables in the set $\{n, switch\}$ must change value in the loop headed by node 6.

At this point we need more information about the paths through the loop headed by node 6. We replace the symbol p_6 by a sub-path-expression which includes details about the body of this loop. We obtain

$$\alpha' = 2 \cdot p_3 \cdot 5 \cdot (6 \cdot 7 \cdot p_8 \cdot 13)^* 6 \cdot 14.$$

We think of α' as adding the information that any path through the loop headed by node 6 consists of zero or more subpaths which start at node 6, go to node 7, take an arbitrary path through the loop headed by node 8, and end at node 13; followed by a final visit to node 6.

Since we are looking for def-clear paths with respect to *switch* we can eliminate any path along which *switch* is always redefined. Since *switch* has a definition in node 7, the only definition clear paths with respect to *switch* are those represented by the path expression

$$\alpha'' = 2 \cdot p_3 \cdot 5 \cdot 6 \cdot 14$$

α'' says that the only path through the loop headed by node 6 which we need to consider is the path consisting of the single node 6. But none of the variables in $\{n, switch\}$ are defined in node 6, E cannot change value in the loop, so the definition-use association is unexecutable.

Overview of the path expression method

We now give a rough overview of the path expression method. The ideas presented here will be made more precise in the following sections.

Given an association (d,u,v) we begin by deriving a path expression α which represents the set of paths from d to u, and which omits details about paths through loops. The path expression has *node symbols* which represent single nodes and *loop symbols* which represent sets of paths through loops. As the algorithm progresses we use a combination of symbolic evaluation and data flow analysis to eliminate from consideration paths which can be shown to be unexecutable or which can be shown to have definitions of v.

For each loop symbol occurring in α we use symbolic evaluation to attempt to find a collection of sets of variables, $\{S_1,...,S_k\}$ such that for each h, at least one of the variables in the set S_h must be redefined within the loop on any *executable* path covering (d,u,v). We then replace the loop symbol by a sub-expression α' which includes more details about the body of the loop. α' represents the set of definition-clear paths with respect to v through the loop. If, for some h, none of the variables in S_h have definitions on any of the paths represented by α' then there are no executable paths in the set of paths represented by α, so the association is unexecutable.

The algorithm terminates when either

1. the current path expression represents the empty set of paths (in this case the association is unexecutable).

or

2. the current path expression contains no more loop symbols (in this case the association may be executable).

3. Definition of Path Expressions

We now define the syntax and semantics of path expressions*. Let P be a procedure with flow graph G.

Let $\Sigma = \{\mathbf{n}_i \mid n_i \in \text{Nodes(G)}\} \cup \{\mathbf{p}_i \mid n_i \text{ is a loop header node in G}\}$. We will refer to the symbols $\mathbf{n}_i$ as *node symbols* and the symbols $\mathbf{p}_i$ as *loop symbols*. We will say that the symbols $\mathbf{p}_i$ and $\mathbf{n}_i$ *correspond* to the node n_i. Let $\Omega = \{\ (,\ ),\ *,\ \cdot,\ \cup,\ \lambda,\ \varnothing\ \}$. A *path expression* is a regular expression $\in (\Sigma \cup \Omega)^*$ defined in the usual way [DAV83]. Specifically,

1. λ is a path expression.

2. $\varnothing$ is a path expression.

3. Each element of Σ is a path expression.

4. If α and β are path expressions then so are

i. $(\alpha \cdot \beta)$

ii. $(\alpha \cup \beta)$

iii. α^*

* Tarjan [TAR81A,TAR81B] and Wegman [WEG83], and others, have also used regular expressions to represent sets of paths through flow graphs. Our formalism differs somewhat from theirs, in that we exploit the structure of the flow graphs, placing special emphasis on loop headers. By so doing, we loose some generality, but gain the ability to ignore details about the internal structure of loops until they are needed.

5. Only those strings formed by a finite number of applications of rules 1 through 4 are path expressions.

We associate a set PATHS(α) of sequences of nodes with each path expression α as follows:

1. PATHS($\boldsymbol{\lambda}$) = $\{\lambda\}$, the set whose only element is the null path.

2. PATHS($\boldsymbol{\varnothing}$) = $\varnothing$, the empty set.

3. PATHS($\mathbf{n}_i$) = $\{n_i\}$, the set whose only element is the path of length one whose only node is n_i.

4. PATHS($\mathbf{p}_i$) = $\{\alpha \mid \alpha$ is a path which begins and ends with node n_i and which contains no nodes which are not in the loop headed by $n_i\}$

5. PATHS($\alpha\cdot\beta$) = PATHS(α)$\cdot$PATHS(β) = $\{\sigma\rho \mid \sigma\in$ PATHS(α) and $\rho\in$ PATHS(β)$\}$

6. PATHS($\alpha \bigcup \beta$) = PATHS(α) $\cup$ PATHS(β)

7. PATHS(α^{*}) = (PATHS(α))*

If S is a set of paths and α is a path expression, we will say that α *represents* S if and only if PATHS(α)=S.

A path expression α is *valid* if every element of PATHS(α) is actually a path in G. That is, α is valid if and only if

$x_1 \cdot \ldots \cdot x_k \in$ PATHS(α) $\Rightarrow$ (x_i, x_{i+1}) is an edge in G for $1 \le i \le k-1$.

We will say that two valid path expressions, α and β are *equivalent*, denoted $\alpha \equiv \beta$ if and only if PATHS(α) = PATHS(β).

As a notational convenience we eliminate extraneous parentheses, adopting the convention that "*" has precedence over "$\cdot$" which has precedence over "$\bigcup$". We will sometimes use the numeral "i" written in bold-face in place of $\mathbf{n}_i$.

We adopt the notational convention of using lower case Greek letters near the beginning of the alphabet to represent path expressions, and lower case Greek letters near the end of the alphabet to represent paths.

If α is a substring of β and α is a path expression we will say that α is a *sub-path-expression* of β.

It will be necessary to distinguish between *sub-path-expressions* and *occurrences of sub-path-expressions*. For the sake of clarity, we will represent path expressions as binary trees in the obvious way. We will refer to the nodes of these binary trees as *vertices* in order to avoid confusing them with nodes of the procedure's flow graph. Each vertex of the binary tree will be labeled with a symbol from $\Sigma \cup \Omega$. Each leaf vertex will be labelled with a node symbol, a loop symbol, λ or $\varnothing$. Each internal vertex will be labelled with $*$, $\cdot$, or $\cup$. Vertices with the symbols $\cdot$ and $\cup$ will have two children, while vertices with the symbol $*$ will only have a left child.

The path-expression corresponding to a tree T can easily be recovered by performing an in-order traversal of T:

```
function PathExpr(T: tree): path-expression;
begin
    if T is the null tree then return null path-expression
    else
    begin
        sym := label(root(T));
        if sym ∈ {"·","*","∪"} then
            return(CONCAT("(",PathExpr(left-subtree),sym,PathExpr(right-subtree),")"))
        else return(sym)
    end
end;
```

Let T be a tree representing some path expression α. Then each subtree of T corresponds to an occurrence of a sub-path-expression of α. We will sometimes denote a subtree of T which corresponds to a sub-path-expression β by T^{β}. The notation $(T^{\alpha}\cdot T^{\beta})$ will denote a tree whose root is labelled with the symbol $\cdot$, whose left subtree is T^{α} and whose right subtree is T^{β}. Similarly, $(T^{\alpha}\cup T^{\beta})$ will denote a tree whose root is labelled with the symbol $\cup$, whose left subtree is T^{α} and whose right subtree is T^{β}; the notation $(T^{\alpha})^{*}$ will denote a tree whose root is labelled with the symbol $*$, whose left subtree is T^{α} and whose right subtree is null. We will sometimes omit parentheses when it is unecessary to distinguish between details of the structure of the trees. For example, $T^{\alpha}\cdot T^{\beta}\cdot T^{\gamma}$ will denote either $((T^{\alpha}\cdot T^{\beta})\cdot T^{\gamma})$ or $(T^{\alpha}\cdot(T^{\beta}\cdot T^{\gamma}))$. The symbol $T^{\alpha}+T^{\beta}$ will denote the set of trees $\{T^{\alpha}\cdot T^{\beta}, T^{\alpha}\cup T^{\beta}, T^{\beta}\cup T^{\alpha}\}$

Let T^{α} be a subtree of T^{γ}. We will say that π is a T^{α}-*subpath* of ρ if and only if $\pi\in$ PATHS(α), $\rho\in$ PATHS(γ), π is a subpath of ρ and either

1. $T^{\gamma}\ T^{\alpha}+T^{\beta}$, or $T^{\gamma}\in T^{\beta}+T^{\alpha}$, or $T^{\gamma}\in(T^{\alpha})^{*}$ or

2. π is a T^{α}-subpath of σ and σ is a T^{β}-subpath of ρ.

Intuitively, if π is a T^{α}-*subpath* of ρ, π is the portion of ρ which comes from the occurrence of the sub-path-expression α of γ corresponding to the subtree T^{α}.

Let T_1 and T_2 be trees, where T_2 may be a subtree of T_1. We define the T_2-*reduction* of T_1 to be the path expression computed by the following function:

```
function Reduce($T_1$,$T_2$): path-expression;
begin
    if $T_1$ is the null tree then return (null path-expression)
    else
    begin
        sym := label(root($T_1$));
        if sym$\in$ {Node-symbols,Loop-symbols} then return(sym)
        else
        begin
            L := left subtree of $T_1$;
            R := right subtree of $T_1$;
            if sym = "$\cdot$" then
                return(CONCAT(Reduce(L,$T_2$),"$\cdot$",Reduce(R,$T_2$)));
            if sym = "$\cup$" then
                if $T_2$ is a subtree of L then return(CONCAT("$\cdot$",Reduce(L,$T_2$)))
                else if $T_2$ is a subtree of R then return(CONCAT("$\cdot$",Reduce(R,$T_2$)))
                else return(CONCAT(Reduce(L,$T_2$),"$\cup$",Reduce(R,$T_2$)))
            if sym = "*" then
                if $T_2$ is a subtree of L then return(CONCAT("$\cdot$",Reduce(L,$T_2$)))
                else return(CONCAT("(",Reduce(L,$T_2$),")","*"));
        end;
    end;
end;
```

Let γ be the T^β-reduction of T^α where T^β is a subtree of T^α. Then γ is a path expression which represents a subset of PATHS(α) consisting entirely of paths having T^β-subpaths. Furthermore, $\gamma=\delta\cdot\beta\cdot\varepsilon$ where δ and ε are path expressions. Note that if there is no subtree T^ζ of T^α such that T^β is a subtree of T^ζ and the root of T^ζ is labelled with "$\cup$" or "*", then the T^β-reduction of T^α is α, itself.

We associate with each valid path expression α three sets of variables, ALWAYS(α), SOMETIMES(α) and NEVER(α) which indicate the variables which have definitions along all, some, or none of the paths in PATHS(α). We define these sets recursively as follows:

Let VAR_SET be the set of all variables visible to the procedure P.
ALWAYS(λ) = $\varnothing$
SOMETIMES(λ) = $\varnothing$
NEVER(λ) = VAR_SET

ALWAYS(∅) = VAR_SET
SOMETIMES(∅) = ∅
NEVER(∅) = VAR_SET

ALWAYS($\mathbf{n}_i$) = {v ∈ VAR_SET | v has a global definition in node i}
SOMETIMES($\mathbf{n}_i$) = ALWAYS($\mathbf{n}_i$)
NEVER($\mathbf{n}_i$) = {v ∈ VAR_SET | v does not have a global definition in node i}

ALWAYS($\mathbf{p}_i$) = {v ∈ VAR_SET | v has a global definition on every path in PATHS($\mathbf{p}_i$)}
SOMETIMES($\mathbf{p}_i$) = {v ∈ VAR_SET | v has a global definition on some path in PATHS($\mathbf{p}_i$)}
NEVER($\mathbf{p}_i$) = {v ∈ VAR_SET | v has a global definition on no path in PATHS($\mathbf{p}_i$)}

ALWAYS(($\alpha \cdot \beta$)) = ALWAYS(α) ∪ ALWAYS(β)
SOMETIMES(($\alpha \cdot \beta$)) = SOMETIMES(α) ∪ SOMETIMES(β)
NEVER(($\alpha \cdot \beta$)) = NEVER(α) ∩ NEVER(β)

ALWAYS(($\alpha \bigcup \beta$)) = ALWAYS(α) ∩ ALWAYS(β)
SOMETIMES(($\alpha \bigcup \beta$)) = SOMETIMES(α) ∪ SOMETIMES(β)
NEVER(($\alpha \bigcup \beta$)) = NEVER(α) ∩ NEVER(β)

ALWAYS(α*) = ∅
SOMETIMES(α*) = SOMETIMES(α)
NEVER(α*) = NEVER(α)

It is straight forward to show that the following properties hold for all path expressions α which are not equivalent to ∅:

1. ALWAYS(α) ⊆ SOMETIMES(α)

2. NEVER(α) = COMPLEMENT(SOMETIMES(α))

3. $v \in \begin{bmatrix} ALWAYS(\alpha) \\ SOMETIMES(\alpha) \\ NEVER(\alpha) \end{bmatrix}$ if and only if v has a global definition on $\begin{bmatrix} all \\ some \\ no \end{bmatrix}$ $\pi \in$ PATHS(α).

4. If $\alpha \equiv \beta$ then ALWAYS(α)=ALWAYS(β), SOMETIMES(α)=SOMETIMES(β), and NEVER(α)=NEVER(β).

4. Partial Symbolic Evaluation of Path Expressions

Symbolic evaluation techniques aim to derive algebraic expressions summarizing the effect of executing a path or set of paths through a program. Symbolic execution derives an expression representing a single path through a program. Global symbolic evaluation attempts to derive an expression representing an entire program.

Most symbolic execution methods attempt to derive both a *path condition* describing the set of input values which cause the path to be executed, and a *path computation*, describing how the values of variables are transformed when the path is executed. Several symbolic execution systems have been described in the literature and have been implemented [KIN76,HOW78b,CHE79,CLA81,KEM85]. Global symbolic evaluation is a much more difficult technique, the goal of which is the derivation of an expression which summarizes the entire effect of a procedure. Its development has been impeded by serious theoretical and practical obstacles.

In this section, we describe a new symbolic evaluation technique, *partial symbolic evaluation of path expressions*. This method is more general than symbolic execution, yet essentially no more expensive. It can provide some but not all of the information which global symbolic evaluation would provide.

We begin with a formal description of symbolic execution. We then present an example of symbolic execution. This is followed by a formal description of partial symbolic evaluation of path expressions followed by an example. Finally, we give a brief description of global symbolic evaluation and compare the three techniques. Initially we will assume that there are no procedure or function calls along any of the paths being symbolically evaluated. We will relax this assumption later.

The systems described in the literature symbolically execute paths from the program's entry to its exit. Our heuristic method for attempting to determine whether a definition-use association is executable involves deriving path conditions and path computations for arbitrary paths. We generalize symbolic execution in a straight-forward way to allow for this.

Let P be a procedure and let $V_1,...,V_n$ be the program variables which are visible to P (i.e., local variables, formal parameters and non-local variables). Let $T_1,...,T_n$ be the types of $V_1,...,V_n$, respectively; let S_i be the set of values which can be assumed by variables of type T_i; and let $U = S_1 \cup ... \cup S_n$. For $1 \le i \le n$, let $S_i' = S_i \cup \{\uparrow\}$ where we think of "$\uparrow$" as denoting the value "undefined". Let $S'_{n+1} = U^*$, the set of finite sequences of elements of U.

We define the *state space of P* to be the set

$$S = S_1' \times ... \times S'_{n+1}.$$

A *state* $v = (v_1,...,v_{n+1})$ is an element of S. If v is a state, we think of $v_1, ... ,v_n$ as representing the values of the variables $V_1, ...,V_n$ and of $v_{n+1} = (X_1,X_2,...)$ as representing the sequence in the input file. For notational convenience we are assuming that P gets its input from a single file *input*. If P took input from k > 1 files, we could let $S'_{n+2},...,S'_{n+k}$=U^* represent these files, and let $S = S_1' \times ... \times S'_{n+k}$.

To each path π in P there corresponds a subset D^π of S called the *path domain of* π defined by

D^π = {$v \in S$ | if at the top of path π the values of variables $V_1,...,V_n$ are $v_1,...,v_n$, respectively, and the sequence of values in the input file is v_{n+1} then path π will be executed} and a function $f^{\pi} = (f_1^{\pi},...,f_{n+1}^{\pi}) : D^{\pi} \rightarrow S$ which transforms an *initial state* to a *final state.*

The goal of symbolic execution is the derivation of a predicate $PC^{\pi}(\mathbf{v})$, called the *path condition*, such that

$PC^{\pi}(\mathbf{v})$ holds if and only if $\mathbf{v} \in D^{\pi}$

and a predicate $COMP^{\pi}(\mathbf{v},\mathbf{w}) \equiv COMP_1^{\pi}(\mathbf{v},\mathbf{w}) \& \ ... \ \& COMP_n^{\pi}(\mathbf{v},\mathbf{w})$, called the *path computation*, such that

$COMP_i^{\pi}(\mathbf{v},\mathbf{w})$ holds if and only if $w_i = f_i^{\pi}(\mathbf{v})$

Thus, the path condition of π describes the path domain of π and the path computation describes the way the values of variables are transformed when π is executed.

Note that if $PC^{\pi} \equiv$FALSE then D^{π} is empty, so π is unexecutable. However the converse does not hold. Recall that a path is executable if it is a subpath of a complete executable path. Thus, if the only *complete paths* having π as a subpath are unexecutable, then π is unexecutable, even if PC^{π} is satisfiable. If π is a complete path, then π is executable if and only if PC^{π} is satisfiable if and only if D^{π} is non-empty.

We now define the predicates $COMP^{\pi}$ and PC^{π}. We will first define $COMP^{\pi}$ and PC^{π} when π consists of at most one node. We will define $COMP^{\pi\rho}$ in terms of $COMP^{\pi}$ and $COMP^{\rho}$ and define $PC^{\pi\rho}$ in terms of PC^{π}, PC^{ρ} and $COMP^{\pi}$.

Let $\pi = \lambda$, the null path. Then $PC^{\pi}(\mathbf{v}) \equiv$ TRUE and $COMP^{\pi}(\mathbf{v},\mathbf{w}) \equiv (\mathbf{v} = \mathbf{w})$.

Let $\pi = (n_1)$ be a path consisting of a single node. Then $PC^{\pi} \equiv$ TRUE and $COMP^{\pi}(\mathbf{v},\mathbf{w})$: is the predicate computed by the following algorithm:

```
type Symbolic_Value : character_string;
    Symbolic_Sequence : sequence of characters;
    Symbolic_State :
                record
```

```
                    v: Symbolic_Value;
                    in: Symbolic_Sequence;
                end;
```

{To avoid confusion we will refer to variables used in this algorithm as "variables" and to variables occurring in the procedure which is being symbolically executed as "program variables". Notation: we will denote variables of type Symbolic_State by bold-face letters and their fields by subscripted letters. For example if **x** is a variable of type Symbolic_State then $x_1,...,x_n$ stand for **x**.v[1],...,**x**.v[n] which represent the symbolic values of the program variables $V_1,...,V_n$; x_{n+1} stands for **x**.in which represents the symbolic value of the input file. We assume that the elements of the sequence x_{n+1} are distinct from one another.}

```
function Symbolically_Execute_Node (n: Node;v,w:Symbolic_State): Predicate;
{Returns an expression representing the predicate $COMP^{\pi}$(v,w)}
var x : Symbolic_State;
begin
    for i := 1 to n+1 do $x_i$ := "$v_i$";
    for sc := 1 to (number of statements in block) do
    begin
        stmt := $sc^{th}$ statement;
        Symbolically_Execute_Statement(stmt,x);
    end;
    for i := 1 to n do
        $COMP_i^{\pi}$ := CONCAT("$w_i$ ", "=", $x_i$);
    $COMP^{\pi}$ := $COMP_1^{\pi}$&...&$COMP_n^{\pi}$
    return($COMP^{\pi}$);
end; {Symbolically_Execute_Node}

procedure Symbolically_Execute_Statement (stmt : StatementType; var x : Symbolic_State);
begin

    if stmt is an input statement,
        "read($V_{i_1}$,...,$V_{i_h}$)" then
    begin
        $x_{i_1}$   := Head($x_{n+1}$);
        $x_{n+1}$ := Tail($x_{n+1}$);
        ...
        $x_{i_h}$   := Head($x_{n+1}$);
        $x_{n+1}$ := Tail($x_{n+1}$);
    end;
    if stmt is an assignment statement,
        "$V_i$ := expr"
    then  $x_i$  := Symbolically-Evaluate-Expr(expr,x);
end; {Symbolically-Evaluate-Statement}

function Symbolically-Evaluate-Expr(expr,x): character_string;
{Returns the string formed by replacing each occurrence of program variable $V_i$ in expr
```

by the value of variable x_i. For example, if *expr* is "V_1+V_2" and the values of x_1 and x_2 are "v_3*v_4" and "5", respectively, then Symbolically-Evaluate-Expr returns the character string "v_3*v_4+5". This could be defined formally in terms of functions Symbolically_Execute_Term and Symbolically_Execute_Factor and a parse of the expression. }

Let $\sigma = \pi\rho$ be the concatenation of the paths π and ρ. The function f^σ corresponding to σ is the composition of the functions f^π and f^ρ corresponding to the paths π and ρ. Thus,

$$COMP^\sigma(\mathbf{v},\mathbf{w}) \equiv COMP^\pi(\mathbf{v},\mathbf{v}') \;\&\; COMP^\rho(\mathbf{v}',\mathbf{w}).$$

Let $BC^{\pi\rho}(V_1,...,V_n)$ be the branch condition on the edge between the last node of π and the first node of ρ. Then

$$PC^\sigma(\mathbf{v}) \equiv PC^\pi(\mathbf{v}) \;\&\; COMP^\pi(\mathbf{v},\mathbf{v}') \;\&\; PC^\rho(\mathbf{v}') \;\&\; BC^{\pi\rho}(\mathbf{v}').$$

It is clear that the predicates $COMP^\pi$ and PC^π satisfy the desired properties.

We can summarize all of the information obtained from symbolically executing a path π in the predicate $SE^\pi(\mathbf{v},\mathbf{w})$ given by

$$SE^\pi \equiv PC^\pi \;\&\; COMP^\pi.$$

Thus, $(\mathbf{v},\mathbf{w})$ satisfies $SE^\pi(\mathbf{v},\mathbf{w})$ if and only if $\mathbf{v} \in D^\pi$ and f^π transforms $\mathbf{v}$ into $\mathbf{w}$.

These recursive definitions of $COMP^\pi$ and PC^π lead in an obvious way to a non-deterministic recursive algorithm for computing the predicates:

```
if length($\pi$) = 0 then $COMP^\pi$:=$COMP^\lambda$
else if length($\pi$) = 1 then
     compute $COMP^\pi$ by above algorithm
else
begin
     let $\pi$=$\sigma\rho$ where length($\sigma$),length($\rho$)$\geq$1;
     compute $COMP^\pi$ from $COMP^\sigma$ and $COMP^\rho$ according to the definition
end;
```

Specifying a particular way to decompose π into σ and ρ gives a deterministic algorithm.

If on each recursive call π is decomposed in such a way that length(σ)=1 the algorithm which results is the *forward expansion* algorithm described in the literature. If π is decomposed in such a way that length(ρ)=1 the algorithm which results is similar to the the *backward expansion* algorithm described in the literature.

We now present an example of symbolic execution. Consider the bubblesort program shown in Figure 5.2.2. We will denote the initial values of the variables *i, a, n, tmp, switch*, and *input* by i^0, a^0, n^0, tmp^0, $switch^0$, and $input^0$ respectively, and the final values by i^f, a^f, n^f, tmp^f, $switch^f$, and $input^f$ respectively. Thus we will be trying to derive the expression

$$SE^{\pi}(\, i^0, a^0, n^0, tmp^0, switch^0, input^0,\ i^f, a^f, n^f, tmp^f, switch^f, input^f)$$

Let π be the path (6,7,8,9,10,12,8,13,6,14). We initialize the path condition to TRUE. Figure 5.4.1 shows the symbolic values of the variables and the path condition after symbolic execution of each subpath. We have only shown those variables which are changed by f^{π} or which are involved in PC^{π}. The entry in the row for subpath ρn_i in the column labelled BC is the predicate $BC^{\rho n_i}(w)$ & $COMP^{\rho}(v^0, w)$.

To obtain $COMP^{\pi}$ we set the final values of variables equal to the symbolic values. To obtain the path condition we conjoin the predicates BC. Conjoining $COMP^{\pi}$ and PC^{π} gives

$$SE^{\pi}(\, i^0, a^0, n^0, tmp^0, switch^0, input^0,\ i^f, a^f, n^f, tmp^f, switch^f, input^f) \equiv$$
$$(a^0[1]>a^0[2]) \ \&\ (n^0=2) \ \&\ (switch^0) \ \&\ (a^f[1]=a^0[2]) \ \&\ (a^f[2]=a^0[1]) \ \&$$
$$(a^f[3]=a^0[3]) \ \&\ (a^f[4]=a^0[4]) \ \&\ (a^f[5]=a^0[5]) \ \&\ (i^f=2) \ \&\ (n^f=1) \ \&\ (switch^f) \ \&$$
$$(input^f=input^0)$$

Thus, symbolic execution of π shows that in order for π to be executed, the value of

n must be equal to 2, the value of *switch* must be TRUE, and the value of a[1] must be greater than the value of a[2] at the beginning of the path. After executing such a path, the value of i will be 2, the value of a[1] and a[2] will have been swapped, and the value of n will be 1.

subpath	a[1]	a[2]	i	n	switch	BC
initial	$a^0[1]$	$a^0[2]$	i^0	n^0	$switch^0$	TRUE
(6)						
(6,7)			1		FALSE	$n^0 \geq 2$ & $switch^0$
(6,7,8)						
(6,7,8,9)						$1 \geq n^0 - 1$
(6,7,8,9,10)	$a^0[2]$	$a^0[1]$			TRUE	$a^0[1] > a^0[2]$
(6,7,8,9,10,12)			2			
(6,7,8,9,10,12,8)						
(6,7,8,9,10,12,8,13)				n^0-1		$2 > n^0 - 1$
(6,7,8,9,10,12,8,13,6)						
(6,7,8,9,10,12,8,13,6,14)						$n^0-1<2$ or not TRUE

Figure 5.4.1

We now describe partial evaluation of path expressions. Let α be a valid path expression. The goal of partial symbolic evaluation of α is the derivation of the predicates $PCOMP^{\alpha}$(v,w), the *partial path computation*, and PPC^{α}(v), the *partial path condition*. Intuitively, PPC will partially describe the common features of the path domains of paths in PATHS(α) and PCOMP will partially describe the common features of path computations corresponding to paths in PATHS(α).

Let β be a sub-path-expression of α consisting only of node symbols, concatenation symbols, lambdas and parentheses. Then PATHS(β) consists of a single path, π. In this case, partial symbolic evaluation of β will be essentially the same as symbolic execution of π. On the other hand, if β contains loop symbols, union symbols, or stars, then PATHS(β) will have more than one element. Unlike global symbolic evaluation, partial symbolic evaluation will not attempt to consider all of the details of each of the (possibly infinitely many) elements of PATHS(β). Rather, it will assign a *new* symbolic value to any variable which has a global definition on at least one path in PATHS(β). Intuitively, if there is more than one way in which the value of a variable can be changed by the functions corresponding to PATHS(β), then the value of that variable is unknown after partially symbolically evaluating β. A precise definition of partial symbolic evaluation follows.

We define $PCOMP^{\alpha}(\mathbf{v},\mathbf{w})$ recursively as follows:

1. If $\alpha = \mathbf{n}_i$ or $\alpha = \boldsymbol{\lambda}$ then $PCOMP^{\alpha}(\mathbf{v},\mathbf{w}) \equiv COMP^{\pi}(\mathbf{v},\mathbf{w})$ where π is the unique element of PATHS(α).

2. If $\alpha = \varnothing$ then $PCOMP^{\alpha}(\mathbf{v},\mathbf{w}) \equiv$ FALSE.

3. For $\alpha \in \{\mathbf{p}_i, (\beta \cup \gamma), \beta^*\}$

   Let SOMETIMES(α) = $\{V_{i_1},\ldots,V_{i_h}\}$. Then

$$PCOMP_i^{\alpha}(\mathbf{v},\mathbf{w}) \equiv \begin{cases} w_i = UNIQUE\_NEW\_SYMBOL & \text{if } i \in \{i_1, \ldots, i_h\} \\ w_i = v_i & \textit{otherwise} \end{cases}$$

4. If $\alpha = (\beta \cdot \gamma)$ then $PCOMP^{\alpha}(\mathbf{v},\mathbf{w}) \equiv PCOMP^{\beta}(\mathbf{v},\mathbf{v}') \; \& \; PCOMP^{\gamma}(\mathbf{v}',\mathbf{w})$.

The definition of $PPC^{\alpha}(\mathbf{v})$ for a path expression α is similar to the definition of $PC^{\pi}(\mathbf{v},\mathbf{w})$ for a path π.

1. If α is a single symbol $\mathbf{n}_i, \mathbf{p}_i, \boldsymbol{\lambda}$, or $\alpha = (\beta \cup \gamma)$ or $\alpha = \beta^*$ then

   $PPC^{\alpha}(\mathbf{v}) \equiv$ TRUE.

2. If α is $\varnothing$ then $PPC^{\alpha}(\mathbf{v}) \equiv$ FALSE.

3. If α is $(\beta \cdot \gamma)$ then

$PPC^{\alpha}(v) \equiv PPC^{\beta}(v)\ \&\ PCOMP^{\beta}(v,v')\ \&\ PPC^{\gamma}(v')\ \&\ BC^{\beta\gamma}(v')$;

We summarize all of the information obtained from partially symbolically evaluating a path expression α with the predicate $PSE^{\alpha}(\mathbf{v},\mathbf{w})$ given by

$$PSE^{\alpha} \equiv PPC^{\alpha}\ \&\ PCOMP^{\alpha}.$$

We now present an example of partial symbolic evaluation. Consider again the bubblesort program shown in Figure 5.2.2. Let α be the path expression

$$\mathbf{6 \cdot 7 \cdot p_8 \cdot 13 \cdot 6 \cdot 14}.$$

We initialize the symbolic values of the variables i, a, n, and *switch* to i^0, a^0, n^0, and $switch^0$, respectively, and we initialize the path condition to TRUE. Figure 5.4.2 shows the symbolic values of the variables and the path condition after executing each subpath. Note that in symbolically executing the loop symbol $\mathbf{p_8}$ we have given new symbolic values to the variables a, switch, and i because each of these variables is potentially redefined in the loop headed by node 8.

Setting the final values of variables equal to the symbolic values, conjoining the entries in the column labelled BC, and simplifying, we get

$PSE^{\alpha}(i^0,a^0,n^0,switch^0,input^0,\ i^f,a^f,n^f,switch^f,input^f) \equiv$

$(a^f{=}a')\ \&\ (i^f{=}i')\ \&\ (n^f{=}n^0{-}1)\ \&\ (switch^f{=}switch')\ \&\ n^0{\ge}2\ \&\ switch^0\ \&\ i'{>}n^0{-}1$

$\&\ (n^0{-}1{<}2 \vee \text{not } switch')$

$\equiv n^0{\ge}2\ \&\ switch^0\ \&\ i^f > n^0{-}1\ \&\ (n^0{=}3 \vee \text{not } switch^f)$

Thus, partial symbolic evaluation of α tells us that in order for a path in PATHS(α) to be executed, the value of n must be greater than or equal to 2 and the value of *switch* must be TRUE at the beginning of the path; after executing such a path, the value of i will be strictly greater than one less than the initial value of n; furthermore, if the initial value of n was not equal to 3, then the final value of switch must be FALSE. This is some, but not all of the information which, in theory, global symbolic evaluation of the set of paths might provide.

subpath	a	i	n	switch	BC
initial	a^0	i^0	n^0	$switch^0$	TRUE
6					
6·7		1		FALSE	$n^0 \geq 2$ & $switch^0$
6·7·p_8	a'	i'		$switch'$	
6·7·p_8·13			n^0-1		$i' > n^0-1$
6·7·p_8·13·6					
6·7·p_8·13·6·14					$(n^0-1 < 2) \vee$ not $switch'$

Figure 5.4.2

For comparison, we will now briefly describe global symbolic evaluation. The descriptions of global symbolic evaluation in the literature deal with global symbolic evaluation of an entire routine. The goal is to derive a single expression which summarizes the effect of executing any path through the routine. It is straight forward to generalize this to global symbolic evaluation of a path expression. Given a path expression α the goal of global symbolic evaluation is the derivation of a predicate $GSE^\alpha(\mathbf{v},\mathbf{w})$ given by

$$GSE^\alpha \equiv \bigvee_{\pi \in PATHS(\alpha)} SE^\pi .$$

$(\mathbf{v},\mathbf{w})$ satisfies $GSE^\alpha(\mathbf{v},\mathbf{w})$ if and only if for some $\pi \in$ PATHS(α), $\mathbf{v} \in D^\pi$ and f^π transforms $\mathbf{v}$ into $\mathbf{w}$. Thus GSE^α gives a complete description of all of the paths in PATHS(α).

Unfortunately, and not surprisingly, there are serious obstacles to automating global symbolic evaluation. If PATHS(α) = $\{\pi_1,\ldots,\pi_n\}$ is finite then it is (at least in principle) possible to automatically derive GSE^α. In this case GSE^α will be of the form

$$(PC^{\pi_1} \rightarrow COMP^{\pi_1}) \,\&\ldots\&\, (PC^{\pi_n} \rightarrow COMP^{\pi_n})$$

The length of this expression may be exponential in the length of α.

The situation is even worse if α contains loop symbols or stars. In this case, PATHS(α) is infinite and it may be impossible to automatically derive GSE^{α}. Global symbolic evaluation methods described in the literature attempt to derive and solve recurrence relations describing the effects of loops. However, it is not always possible to do so. Nested loops and selection statements within loops are particularly problematic.

Consider the path expression α=**6·7·p$_8$·13·6·14** to which we applied partial symbolic evaluation above. Manually analyzing the set of paths represented by α shows that

$GSE^{\alpha}(\mathrm{v,w}) \equiv$

$$(n^0 \geq 2) \;\&\; switch^0 \;\&$$
$$((n^0 > 2) \rightarrow ((a^0[n^0-1] \leq a^0[n^0]) \;\&\; (a^f[n^0]=a^0[n^0])))$$
$$\&\; (\forall \mathrm{j})((1 \leq \mathrm{j} < n^0) \rightarrow (a^f[n^0] \geq a^f[\mathrm{j}]))$$
$$\&\; i^f=n^0 \;\&\; n^f=n^0-1.$$

This expression gives more detail about the effect of the paths through the loop headed by node 6. Unfortunately, however, the global symbolic evaluation methods described in the literature are not powerful enough to derive GSE^{α} for this path expression.

The following facts about the relationship between the symbolic execution, partial symbolic evaluation, and global symbolic evaluation are easy to verify:

1. For each $\pi \in$ PATHS(α). $SE^{\pi} \Rightarrow PSE^{\alpha}$

2. $GSE^{\alpha} \Rightarrow PSE^{\alpha}$

3. If PATHS(α)=$\{\pi\}$ then $SE^{\pi} \equiv PSE^{\alpha} \equiv GSE^{\alpha}$

When PATHS(α) consists of a single path, the three methods are equivalent. Global symbolic evaluation is more powerful than partial symbolic evaluation, but partial symbolic evaluation is much more practical, being essentially no more difficult than symbolic execution.

We conclude this section with some comments on procedure calls. Paths containing procedure calls are problematic for symbolic execution systems. (Note that since we are assuming that functions cannot take **var** parameters or have side effects, they do not present the same problems as procedure calls do.) Let π be a path consisting of a single node whose only statement is a procedure call. One would like $SE^{\pi}(\mathbf{v},\mathbf{w})$ to hold if and only if when the procedure is called with initial state $\mathbf{v}$ the procedure terminates and the state upon returning from the call is $\mathbf{w}$. However, since there is no way to determine *a priori* the path which will be followed through the procedure, in order to derive $SE^{\pi}(\mathbf{v},\mathbf{w})$ one would have to employ global symbolic evaluation.

One approach which has been suggested for symbolically executing paths with procedure calls is the replacement of the node corresponding to the procedure call by a particular path through the called procedure. This is equivalent to disallowing procedure calls, and seems overly restrictive. Another approach is to replace the procedure call by an executable specification for the procedure. This seems more reasonable, but it assumes the availability of an executable specification for each procedure and assumes that the procedure correctly implements that specification.

For partial symbolic evaluation, on the other hand, there is an easy way to treat nodes having procedure calls. The approach is consistent with the treatment of other symbols corresponding to more than one path. If α is a node whose only statement is a procedure call then we simply define

$$PCOMP_i^{\alpha}(v,w) \equiv \begin{cases} w_i = UNIQUE\_NEW\_SYMBOL & \text{if } i \in \{i_1, ..., i_h\} \\ w_i = v_i & \textit{otherwise} \end{cases}$$

where $V_{i_1}, ..., V_{i_h}$ are the variables which can potentially be defined by the procedure call (actual parameters corresponding to **var** formal parameter and global variables which are modified in the called procedure).

5. Precise Description of the Path Expression Method

In this section we give a precise description of the path expression method. Let (d,u,v) be definition-use association. The path expression method will attempt to determine that (d,u,v) is unexecutable, that is, that there is no executable path from the procedure's entry to its exit which has as a subpath a definition-clear path with respect to v from d to u.

We can assume, without loss of generality, that u is a node, rather than an edge. This is because we have defined structured flow graphs in such a way as to insure that if (i,j) is an edge having a p-use of some variable, then there is no edge (i,k) for $k \neq j$. Thus, if (d,(i,j),v) is a definition-p-use association, the paths from d to (i,j) are precisely the paths from d to j.

A *candidate* is a path from the procedure's entry to its exit which has a subpath going from d to u. A candidate is *viable* if it is executable and if it has a definition clear subpath with respect to v from d to u. Thus, the definition-use association is executable if and only if the set of viable candidates is non-empty.

We will use a path expression ξ to describe the set of candidates. As the path expression method progresses, we will attempt to eliminate from consideration subsets which consist only of candidates which are not viable. When such a subset is eliminated, we will rewrite ξ accordingly. If we can eliminate all of the candidates from consideration then the association is unexecutable. In this case, ξ will be equivalent to the path expression $\emptyset$.

Let α be a path expression which begins with $\mathbf{n}_j$ or $\mathbf{p}_j$ and let (i,j) be an edge in G. We will sometimes prepend α with a symbol $\mathbf{n}_i^{top}$ or the symbol $\mathbf{p}_i^{top}$. This will indicate that we are including the branch condition on edge (i,j) in the partial symbolic evaluation of α. Similarly, when α ends with $\mathbf{n}_j$ or $\mathbf{p}_j$ and (j,i) is an edge in G we will sometimes append α with $\mathbf{n}_i^{bot}$ or $\mathbf{p}_i^{bot}$. This will indicate that we are including the branch condition on edge (j,i) in the partial symbolic evaluation of α.

Let $T^{\alpha^{1,d}}$ be the tree corresponding to a path expression representing the set of paths from the entry node to the bottom of node d. Let $T^{\alpha^{d,u}}$ be the tree corresponding to a path expression representing the set of paths which go from the bottom of node d to the top of node u without passing through d or u. Let $T^{\alpha^{u,n}}$ be the tree corresponding to a path expression representing the set of paths from the top of node u to the exit node. Let $T^{\xi} \in T^{\alpha^{1,d}} \cdot T^{\alpha^{d,u}} \cdot T^{\alpha^{u,n}}$. Then every candidate is an element of PATHS(ξ) of the form $\pi\rho\sigma$ where $\pi \in$ PATHS($\alpha^{1,d}$), $\rho \in$ PATHS($\alpha^{d,u}$), and $\sigma \in$ PATHS($\alpha^{u,n}$). The viable candidates are executable paths of this form where ρ is a definition-clear path with respect to v. Throughout this section, T^{ξ} will stand for the tree representing the current set of candidates and $T^{\alpha^{d,u}}$ will stand for the current subtree corresponding to the definition-clear subpaths with respect to v from d to u.

We begin this section by defining a set of rules which will be used to rewrite path expressions. We will then define some notions which will be used for showing that certain sets of paths contain no executable paths. We then present two path-expression-method algorithms. The first, PEM1, is a very general non-deterministic procedure which is not guaranteed to terminate. The second, PEM2, is an efficient but less general algorithm, in which most of the non-determinism has been removed. We conclude with a discussion of the implementation of PEM2 and two examples.

We now define a set of rewrite rules which will be used in the path expression method:

RW1:Replace a subtree T^{α} of T^{ξ} by $T^{\varnothing}$.

RW2:Replace a subtree T^{α} of T^{ξ} by $T^{\alpha'}$ where $\alpha \equiv \alpha'$.

Let T^{α} be a subtree of T^{ξ}. If $T^{\xi'}$ is the tree obtained by applying RW2 to T^{α}, i.e. by replacing the subtree T^{α} by $T^{\alpha'}$, then PATHS(ξ')=PATHS(ξ). So application of RW2 results in a different representation of the same set of candidate paths. On the other hand, if $T^{\xi'}$ is the tree obtained by applying RW1 to T^{α}, then PATHS(ξ')$\subseteq$PATHS(ξ). Before we apply rule RW1, we will have to insure that none of the paths in PATHS(ξ)−PATHS(ξ') is viable.

Let T^{α} be a subtree of T^{ξ}. We will say that T^{α} is *RW1-safe* if and only if no viable candidate in PATHS(ξ) has a T^{α}-subpath.

Let T^{α} be an RW1-safe subtree of T^{ξ}. Let $T^{\xi'}$ be the tree obtained from T^{ξ} by applying RW1 to T^{α}. Then all of the paths in PATHS(ξ)−PATHS(ξ') have T^{α}-subpaths, hence are not viable. So we can safely replace ξ by ξ', thereby eliminating from consideration some of the non-viable candidates.

We now discuss methods for trying to determine whether or not a subtree is RW1-safe. The simplest method of showing a subtree to be RW-safe uses only syntactic information. If T^{α} is a subtree of $T^{\alpha^{d,u}}$ and $v \in$ ALWAYS(α) then no path having a T^{α}-subpath is viable, so T^{α} is RW1-safe.

Other methods of showing a subtree to be RW1-safe take into account the semantics of the procedure. Let E be a predicate whose free variables are some but not necessarily all of the program variables and let α be a path expression. We will say that E is α-*satisfiable* if and only if there exist states **v** and **w** such that $GSE^{\alpha}(\mathbf{v},\mathbf{w})$ & E(**w**). Thus, E is α-satisfiable if and only if for some path $\pi \in$ PATHS(α) and some state v, $\mathrm{v} \in D^{\pi}$ and f^{π} transforms v into w and w satisfies E. In particular, if PATHS(α) is the set of paths from procedure entry to point p then E is α-satisfiable if and only if when control reaches point p, the program state satisfies predicate E. We will say that E is α-*unsatisfiable* if and only if E is not α-satisfiable.

In the path expression method we will be interested in showing that certain predicates are α-unsatisfiable for certain path expressions α. Since it is not in general possible to derive GSE^{α} we will approximate GSE^{α} with a weaker predicate. Let C(v,w) be any predicate such that $GSE^{\alpha} \Rightarrow$C. Clearly, if (v,w) satisfies GSE^{α}(v,w) & E(w) then (v,w) satisfies C(v,w) & E(w). Thus if C(v,w) & E(w) is unsatisfiable then E is α-unsatisfiable. In particular, if PSE^{α}(v,w) & E(w) is unsatisfiable then E is α-unsatisfiable.

We will say that E is α-*necessary* if and only if $(\bigvee_{\pi \in PATHS(\alpha)} PC^{\pi}) \Rightarrow$E. That is, if v is in the path domain of any element of PATHS(α) then v satisfies E. For example, the predicate PPC(α) is α-necessary. It follows from the definitions that if a predicate E≡FALSE is α-necessary then all of the paths in PATHS(α) are unexecutable, hence none of them are subpaths of viable paths, so T^{α} is RW1-safe.

Note that the question of whether or not E is α-satisfiable pertains to program states which can occur *after* executing a path from PATHS(α) while the question of whether or not E is α-necessary pertains to program states which must occur *before* executing a path from PATHS(α).

Let S be a set of program variables and let T^{α} be a subtree of T^{ξ}. We will say that S has the *T^{α}-Must-Redefine* property if and only if at least one element $V \in S$ has a definition on each T^{α}-subpath of a *viable* path in PATHS(ξ). Thus, if S satisfies the T^{α}-Must-Redefine property and $S \subseteq$NEVER(α) then no viable element of PATHS(ξ) has an T^{α}-subpath. In other words, if S satisfies the T^{α}-Must-Redefine property and $S \subseteq$NEVER(α) then T^{α} is RW1-safe.

Let $\alpha = \beta \cdot \gamma \cdot \delta$ be a sub-path-expression of ξ, where β, γ, and δ are sub-path-expressions of α. Let E($V_1, \ldots, V_n$) be a predicate which is both δ-necessary and β-unsatisfiable. Let *VARIABLES(E)* denote the set of program variables which actually occur in E. Let π be an *executable* element of PATHS(α). Then $\pi = \rho\sigma\tau$ where $\rho \in$ PATHS(β), $\sigma \in$ PATHS(γ), and $\tau \in$ PATHS(δ) are all executable paths. Let $v \in D^{\pi}$. f^{ρ} transforms v into a state v′ which *does not* satisfy E (since E is β-unsatisfiable). f^{σ} transforms v′ into a state w which *does* satisfy E (since $w \in D^{\tau}$ and E is δ-necessary). So at least one of the variables in VARIABLES(E) must change value while path σ is being executed. Thus, at least one of the variables in VARIABLES(E) must have a definition on path σ.

Combining the above definitions and observations gives

LEMMA 5.5.1:

Let T^{α} be a subtree of T^{ξ} where $T^{\alpha} \in T^{\beta} \cdot T^{\gamma} \cdot T^{\delta}$. Let E($V_1, \ldots, V_n$) be a predicate which is both δ-necessary and β-unsatisfiable. Then the set VARIABLES(E) satisfies the T^{γ}-Must-Redefine property. If VARIABLES(E)$\subseteq$NEVER(γ) then T^{γ} is RW1-safe.■

We will associate with each subtree T^{α} of T^{ξ} a collection MUST-REDEFINE(T^{α}) of sets of variables, such that each $S \in$ MUST-REDEFINE(T^{α}) satisfies the T^{α}-Must-

Redefine Property. Initially, each of the collections MUST-REDEFINE(T^{α}) will be empty. As the path expression method progresses, we will use Lemma 5.5.1 to derive elements of the collections.

The following observations justify associating the collection MUST-REDEFINE(T^{α}) with the root of T^{α}. Let T be a subtree of T^{α} where T^{α} is a subtree of T^{ξ} and let S be a set of program variables which satisfies the T^{α}-MUST-REDEFINE property. Let $T^{\alpha'}$ and $T^{\xi'}$ be the trees obtained by applying RW2 to T. Then since PATHS(ξ)=PATHS(ξ') and PATHS(α)=PATHS(α'), S satisfies the $T^{\alpha'}$-Must-Redefine property (with respect to ξ'). Similarly, if T^{β} is RW1-safe and if $T^{\alpha'}$ and $T^{\xi'}$ are the trees obtained by applying RW1 to T then S satisfies the $T^{\alpha'}$-Must-Redefine property (with respect to $T^{\xi'}$). Thus if S belongs to the collection MUST-REDEFINE(T^{α}) associated with the root of T^{α}, and if RW2 is applied to a subtree of T^{α} or RW1 is applied to a RW1-safe subtree of T^{α} then S still satisfies the T^{α}-Must-Redefine property for the transformed tree.

We now present a non-deterministic procedure which attempts to determine whether an association is unexecutable. It returns "UNEXECUTABLE" only if the association actually is unexecutable. We will then discuss a strategy for eliminating the non-determinism and increasing the efficiency of the algorithm. An algorithm for computing a path expression representing the set of paths from a node s to a node or edge t is given in Appendix 1.

Algorithm 5.5.1:

```
function PEM1(d,u,v): (POSSIBLY-EXECUTABLE,UNEXECUTABLE);
initialize $T^{\xi}$ and $T^{\alpha^{d,u}}$
for each subtree $T^{\alpha}$ of $T^{\xi}$ do
begin
        MUST-REDEFINE($T^{\beta}$):=$\varnothing$;
        RW1-safe($T^{\beta}$) := false
end;
done:=false;
while not done do
begin
        if $\xi \equiv \varnothing$ then return(UNEXECUTABLE);
        else select

□ true ⇒    Apply RW1 to an RW1-safe subtree $T^{\alpha}$ of $T^{\xi}$;

□ true ⇒    Apply RW2 to a subtree $T^{\alpha}$ of $T^{\xi}$;

□ true ⇒    UPDATE MUST-REDEFINE:
            choose a subtree $T^{\gamma}$ of $T^{\xi}$;
            let $\alpha=\beta\cdot\gamma\cdot\delta$ be the $T^{\gamma}$-reduction of $T^{\xi}$;
            find a $\delta$-necessary predicate E;
            if E is $\beta$-unsatisfiable then
                    MUST-REDEFINE($T^{\gamma}$):=MUST-REDEFINE($T^{\gamma}$)$\cup${VARIABLES(E)};

□ true ⇒    UPDATE RW1-safe:
            Choose a subtree $T^{\alpha}$ of $T^{\alpha^{d,u}}$ such that $v \in$ ALWAYS($\alpha$)
            or choose a subtree $T^{\alpha}$ of $T^{\xi}$ such that MUST-REDEFINE($T^{\alpha}$)$\subseteq$NEVER($\alpha$)
            or choose a subtree $T^{\alpha}$ of $T^{\xi}$ such that FALSE is $\alpha$-necessary;
            RW1-safe($T^{\alpha}$) := true;

□ true ⇒    done:=true; {give up}
        end select;
end;
return(POSSIBLY-EXECUTABLE);
```

As indicated above, application of the rewrite rule RW1 to RW1-safe states does not eliminate any viable candidates. Thus every time the top of the while loop is reached all of the viable candidates belong to PATHS(ξ). If $\xi \equiv \varnothing$ then PATHS(ξ) is empty, so there are no viable candidates. In this case the procedure terminates, returning "UNEXECUTABLE". However, the algorithm is not guaranteed to terminate. If at any point the "done := true" selection is executed the algorithm will terminate returning "POSSIBLY-EXECUTABLE". In this case PATHS(ξ) is a superset of the set of viable paths.

It would be possible to eliminate the non-determinism in PEM1, by selecting a particular order in which to perform the operations and to restrict the application of RW2 so as to avoid the generation of trees which had already been generated at a previous step. However, even with these restrictions it would be possible for ξ to become arbitrarily long, for example by using RW2 to repeatedly replace a sub-path-expression α^* by $((\alpha \cdot \alpha^*) \cup \lambda)$.

In the function PEM2, we will sacrifice some of the generality of PEM1, for the sake of efficiency. We will do this by only allowing a restricted form of the rewrite rule RW2:

RW2′:

> Replace a subtree T^{α} of T^{ξ} by $T^{\alpha'}$ where $\alpha = \mathbf{p}_i$, $\alpha' = (\beta^* \cdot \mathbf{n}_i)$, no node symbol or loop symbol occurs more than once in β, and $\alpha' \equiv \alpha$. (Thus α' represents the set of paths through the loop headed by node i.)

RW2′ transforms ξ into an equivalent path expression which includes more details about the body of some loop.

Function PEM2 uses rewrite rules RW1 and RW2′, and thus concentrates attention on loops in the attempt to eliminate non-viable candidates. PEM2 will also focus attention on the subtree $T^{\alpha^{d}{}_{u}}$, applying the rewrite rules only to subtrees of this tree. PEM2 also eliminates (most of) the non-determinism of PEM1 by performing the operations in a pre-determined order.

Algorithm 5.5.2

```
function PEM2(d,u,v): (POSSIBLY-EXECUTABLE,UNEXECUTABLE);
Initialize $T^{\xi}$ and $T^{\alpha^{d,u}}$;
for each subtree $T^{\alpha}$ of $T^{\xi}$ do MUST-REDEFINE($T^{\alpha}$):=$\varnothing$;
while $T^{\alpha^{d,u}}$ has a subtree of the form $T^{\mathbf{p}_i}$ do
begin
{1}   if $\xi \equiv \varnothing$ then return(UNEXECUTABLE)
      else
      begin
{2}       choose a subtree $T^{\mathbf{p}_i}$ of $T^{\alpha^{d,u}}$;
{3}       let $\alpha$ be the $T^{\mathbf{p}_i}$-reduction of $T^{\xi}$;
          then $\alpha = \beta' \cdot \mathbf{p}_i \cdot \gamma'$ where $\beta'$ and $\gamma'$ are sub-path-expressions;
          let $\beta$ denote $\beta' \cdot \mathbf{p}_i^{top}$ and let $\gamma$ denote $\mathbf{p}_i^{bot} \cdot \gamma'$;
{4}       derive $\gamma$-necessary predicates $E_1,...,E_k$;
{5}       if $E_1 \& ... \& E_k \equiv$FALSE then {FALSE is $\gamma$-necessary} apply RW1 to $T^{\alpha}$
          else begin
              for i := 1 to k do
              begin
{6}               if $E_i$ is $\beta$-unsatisfiable
{7}               then MUST-REDEFINE($T^{\gamma}$)
                  := MUST-REDEFINE($T^{\gamma}$)$\cup${VARIABLES(E)};
              end; {for}
          end; {else}
{8}       Apply RW2' to $T^{\mathbf{p}_i}$;
          {let $T^{\delta}$ denote the subtree which replaces $T^{\mathbf{p}_i}$}
{9}       Apply RW1 to any subtree $T^{\varepsilon}$ of $T^{\delta}$ such that v$\in$ALWAYS($\varepsilon$);
{10}      Update sets ALWAYS($\zeta$), SOMETIMES($\zeta$), NEVER($\zeta$) for all
              sub-path-expressions $\zeta$ of $\xi$.
{11}      Apply RW1 to any subtree $T^{\varepsilon}$ of $T^{\xi}$ such that
              ($\exists$S$\in$MUST-REDEFINE($T^{\varepsilon}$) s.t. S$\subseteq$NEVER($\varepsilon$)).
      end; {else}
end; {while}
if PPC($\xi$)$\equiv$FALSE then {FALSE is $\xi$-necessary} return(UNEXECUTABLE)
else return(POSSIBLY-EXECUTABLE);
```

Each time the top of the while loop in PEM2 is reached, the set of viable candidates is contained in PATHS(ξ). Thus, PEM2 returns "UNEXECUTABLE" only if the definition-use association (d,u,v) is unexecutable. The algorithm terminates either when $\xi \equiv \varnothing$ or when there are no more loop symbols to which to apply RW2'. If PEM2 returns "POSSIBLY-EXECUTABLE" then the final value of ξ represents a super-set of the set of viable candidates.

We will next address a number of implementation issues. We assume the availability of a partial symbolic evaluator and a theorem prover.

In line 1 the algorithm has to determine whether ξ is equivalent to $\varnothing$. One way to make it easier to do this is by performing the following simplifications whenever T^{ξ} is modified:

1. Replace subtrees of the form $T^{\varnothing} \cdot T^{\delta}$ or $T^{\delta} \cdot T^{\varnothing}$ by $T^{\varnothing}$.
2. Replace subtrees of the form $T^{\varnothing} \cup T^{\delta}$ or $T^{\delta} \cup T^{\varnothing}$ by T^{δ}.
3. Replace subtrees of the form $T^{\lambda} \cdot T^{\delta}$ or $T^{\delta} \cdot T^{\lambda}$ by T^{δ}.
4. Replace subtrees of the form $T^{\varnothing *}$ by T^{λ}.
5. Replace subtrees of the form $T^{\lambda *}$ by T^{λ}.

These simplifications can be performed in a single post-order traversal of the tree T^{ξ}. After performing these simplifications, $\xi \equiv \varnothing$ if and only if $\xi = \varnothing$.

Each time execution reaches line 2 the algorithm chooses a subtree, T^{P_i}, on which to concentrate. This non-determinism can be eliminated by specifying a particular order in which loop symbols are to be processed. For example it could be stipulated that loop symbols are to be processed in increasing order of depth of nesting and from left to right within a given depth of nesting. More work is needed to determine how the order in which loop symbols are processed effects the efficiency and power of the algorithm.

In line 4, the algorithm derives a set $\{E_1, \ldots, E_k\}$ of γ-necessary predicates. In order to derive *all* of the γ necessary predicates, we would have to employ global symbolic evaluation of γ. However, we can derive *some* of them by using partial symbolic evaluation. Recall that the the predicate PPC^{γ} is γ-necessary. So we can derive this family by writing PPC^{γ} as a conjunct of clauses, $PPC^{\gamma}(v) \equiv E_1(v) \& \ldots \& E_k(v)$.

Similarly, in line 6, the algorithm must determine whether E_i is β-unsatisfiable, i.e. whether $GSE^{\beta}(\mathrm{v},\mathrm{w})\&E_i(\mathrm{w})$ is unsatisfiable. To get a precise answer to this question we would need to perform global symbolic evaluation to derive the predicate and to employ a sufficiently powerful theorem prover to check its satisfiability. It is not in general possible either to perform global symbolic evaluation or to determine whether an arbitrary predicate is satisfiable. However, we can perform the following approximation. As pointed out earlier, if $PSE^{\beta}(\mathrm{v},\mathrm{w})\&E_i(\mathrm{w})$ is unsatisfiable then E_i is β-unsatisfiable. If the theorem prover can show that $PSE^{\beta}(\mathrm{v},\mathrm{w})\&E_i(\mathrm{w})$ is unsatisfiable then we know that E_i is β-unsatisfiable. If the theorem prover determines that $PSE^{\beta}(\mathrm{v},\mathrm{w})\&E_i(\mathrm{w})$ is satisfiable or if it cannot reach a conclusion (within some predetermined amount of time) then PEM2 can make the conservative assumption that E_i is *not* β-unsatisfiable (although in fact it might be.)

The sets ALWAYS, SOMETIMES, and NEVER can easily be updated (line 10) whenever T^{ξ} is modified by performing a single post-order traversal of the tree, applying the definitions of the sets. At the same time, RW1 can be applied to subtrees which have become RW1-safe by virtue of the updates (line 11).

Note that given a partial symbolic evaluator, a theorem prover, and a strategy for choosing the subtree T^{p_i} on which to concentrate, PEM2 becomes a fully automatic procedure. We now show that the time-complexity of PEM2 is polynomial in the length of the subject program.

The algorithm for initializing $T^{\alpha^{d,u}}$ produces a path expression in which each loop symbol appears at most twice. Let p_i be a loop symbol appearing in $T^{\alpha^{d,u}}$. The loop symbols which are generated by repeatedly applying RW2′ to T^{p_i} are all distinct from one another. So the total number of passes through the while loop is less than or equal to

$2L^2$ where L is the number of loops in the procedure.

We can also guarantee that at any point in the algorithm, the length of ξ is $O(N^2)$ where N is the number of nodes in the flow graph (see Appendix II). Let M be the number of characters in the subject program. Clearly N=O(M) and L=O(M).

In line 1, PEM2 checks whether $\xi \equiv \emptyset$. This can be done in $O(N^2)$ time by simplifying T^{ξ} during one traversal of the tree. There are many possible strategies for choosing a subtree T^{p_i} on which to concentrate in line 2. Some of these strategies take only O(1) time. Finding the T^{p_i}-reduction of T^{ξ}, and the corresponding sub-path-expressions β and ξ in step 3 can be done with one traversal of the tree, hence in $O(N^2)$ time. Partial symbolic evaluation of a path expression γ can be done in time proportional to the length of γ which is $O(N^2)$ so step 4 can be performed in $O(N^2)$ time. Since the number of clauses in each branch is O(M), the number, k, of predicates E_i derived in step 4 is $O(N^2 M)=O(M^3)$.

Since, as discussed above, the theorem prover can be forced to return after some predetermined constant amount of time, each execution of step 5 and of 6 takes O(1) time. The **for** loop is executed $O(M^3)$ times, and each execution takes O(1) time. Steps 8, 9, and 10 can each be performed in $O(N^2)$ time. In step 11 there are $O(N^2)$ subtrees to examine, each of which can potentially have $O(M^3)$ elements in its MUST-REDEFINE set, so step 11 takes $O(N^2 M^3)=O(M^5)$ time.

Thus PEM2 runs in time bounded by $O(L^2 M^5)=O(M^7)$. The above estimate was very crude, and was strongly affected by the fact that $k=O(M^3)$. This is not a tight upper bound on k. Furthermore, in practice, k will generally be a small number, and the algorithm will run more efficiently than this worst-case upper bound would indicate.

It is also possible to implement PEM2 in such a way that only the "relevant" loops need be examined. This improves the run time in practice. To do this, it is necessary to make conservative estimates of the sets ALWAYS($\mathbf{p}_i$), SOMETIMES($\mathbf{p}_i$), NEVER($\mathbf{p}_i$) as follows. Let the definitions of *ALWAYS'*, *SOMETIMES'*, and *NEVER'*, be analogous to those of ALWAYS, SOMETIMES, and NEVER except that *ALWAYS'*($\mathbf{p}_i$)=ALWAYS($\mathbf{n}_i$), *SOMETIMES'*($\mathbf{p}_i$)=VAR_SET; and *NEVER'*($\mathbf{p}_i$)=$\emptyset$. Then for any path expression α, *ALWAYS'*(α)$\subseteq$ALWAYS(α), *SOMETIMES'*(α)$\supseteq$SOMETIMES(α), and *NEVER'*(α)$\subseteq$NEVER(α). Therefore, any subtree which is marked as being RW1-safe, using the sets *ALWAYS'* and *NEVER'* will in fact be RW1-safe. Also, when performing partial symbolic evaluation, any variable which should be given a unique new symbolic value will be given one.

We next present two examples of the operation of PEM2. In the first, we re-examine the bubble-sort procedure presented in section 5.2. In the second we examine a program in which the bodies of deeply nested loops are relevant to determining that an association is unexecutable. When no ambiguity results, we will refer to applying rewrite rules to *sub-path-expressions*, rather than to the corresponding *trees*. Figures 5.5.1 and 5.5.3 show how the trees T^{ξ} and $T^{\alpha^{d,u}}$ evolve as the algorithm proceeds in Examples 5.5.1 and 5.5.2, respectively. In these figures, the root of a subtree T is annotated with a list of the elements of MUST-REDEFINE(T).

Example 5.5.1

Consider again the definition-use association (2,(6,14),*switch*) from the bubble-sort program in Figure 5.2.2. Initializing ξ and $\alpha^{d,u}$ gives $\alpha^{d,u}$=$\mathbf{2}\cdot\mathbf{p}_3\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{14}$ and ξ=$\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{p}_3\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{14}\cdot\mathbf{15}$.

We first direct attention toward the loop symbol $\mathbf{p}_3$. The $T^{\mathbf{p}_3}$-reduction of T^{ξ} is is ξ itself. $\beta=\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{p}_3{}^{top}$ and $\gamma=\mathbf{p}_3{}^{bot}\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{14}\cdot\mathbf{15}$. Partial symbolic evaluation of γ gives $PPC^{\gamma}\equiv E_1\equiv(i>5)$. Partial symbolic evaluation of β gives $PCOMP^{\beta}\Rightarrow(i=1)$ so E_1 is β-unsatisfiable. We place the element VARIABLES$(E_i)=\{i\}$ in the set MUST-REDEFINE$(T^{\mathbf{p}_3})$. (See Figure 5.5.1.a).

We next apply RW2′ to $T^{\mathbf{p}_3}$, replacing $\mathbf{p}_3$ by $\delta=(\mathbf{3}\cdot\mathbf{4})^{*}\cdot\mathbf{3}$. (See Figure 5.5.1.b). There are no opportunities to apply RW1 to δ. Control goes to the top of the while loop with $\alpha^{d,u}=\mathbf{2}\cdot(\mathbf{3}\cdot\mathbf{4})^{*}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{14}$ and $\xi=\mathbf{1}\cdot\mathbf{2}\cdot(\mathbf{3}\cdot\mathbf{4})^{*}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{14}\cdot\mathbf{15}$.

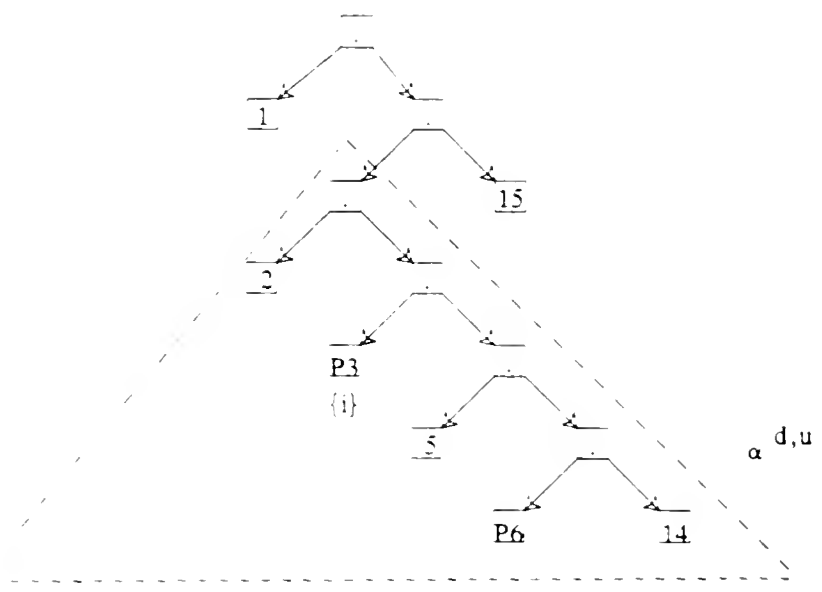
We next concentrate on loop symbol $\mathbf{p}_6$. The $T^{\mathbf{p}_3}$-reduction of T^{ξ} is is ξ itself. $\beta=\mathbf{1}\cdot\mathbf{2}\cdot(\mathbf{3}\cdot\mathbf{4})^{*}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{p}_6{}^{top}$ and $\gamma=\mathbf{p}_3{}^{bot}\cdot\mathbf{14}\cdot\mathbf{15}$. Partial symbolic evaluation gives $PPC^{\gamma}\equiv E_1\equiv((n<2)$ or (not $switch$)) and $PCOMP^{\beta}\Rightarrow(switch\&(n=5))$, so E_1 is β-unsatisfiable. The set VARIABLES$(E_1)=\{n,switch\}$ is placed in MUST-REDEFINE$(T^{\mathbf{p}_6})$. We apply RW2′, replacing $\mathbf{p}_6$ by $\delta=(\mathbf{6}\cdot\mathbf{7}\cdot\mathbf{p}_8\cdot\mathbf{13})^{*}\cdot\mathbf{6}$. (See Figure 5.5.1.c). Since $switch\in$ ALWAYS$((\mathbf{6}\cdot\mathbf{7}\cdot\mathbf{p}_8\cdot\mathbf{13}))$ we can apply RW1 to this sub-path-expression. (See Figure 5.5.1.d). After simplifying, we have $\alpha^{d,u}=\mathbf{2}\cdot(\mathbf{3}\cdot\mathbf{4})^{*}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{6}\cdot\mathbf{14}$ where $\{n,switch\}\in$ MUST-REDEFINE(T^{6}). Since $\{n,switch\}\subseteq$NEVER(6) we can apply RW1 to 6. (See Figure 5.5.1.e).

Simplifying, gives $\xi=\varnothing$. (See Figure 5.5.1.f). Control returns to the top of the while loop, and, since there are no remaining loop symbols, exits from the loop. Since $PPC^{\xi}=PPC^{\varnothing}\equiv$FALSE, PEM2 returns "UNEXECUTABLE".

Similarly, if we had concentrated attention on $\mathbf{p}_6$ first, PEM2 would have returned "UNEXECUTABLE". In fact, in this case it would not have had to apply RW2′ to $T^{\mathbf{p}_3}$.

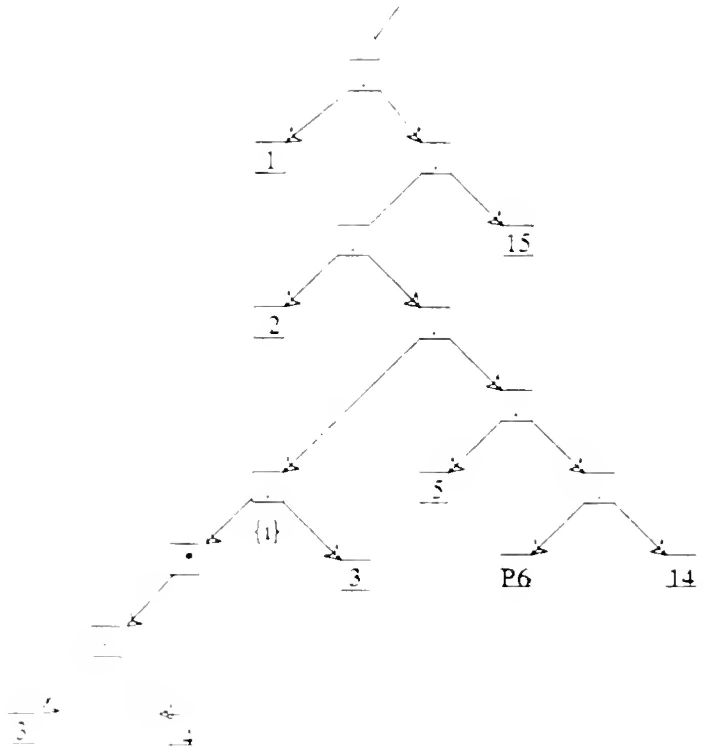
FIGURE 5.5.1

(a)



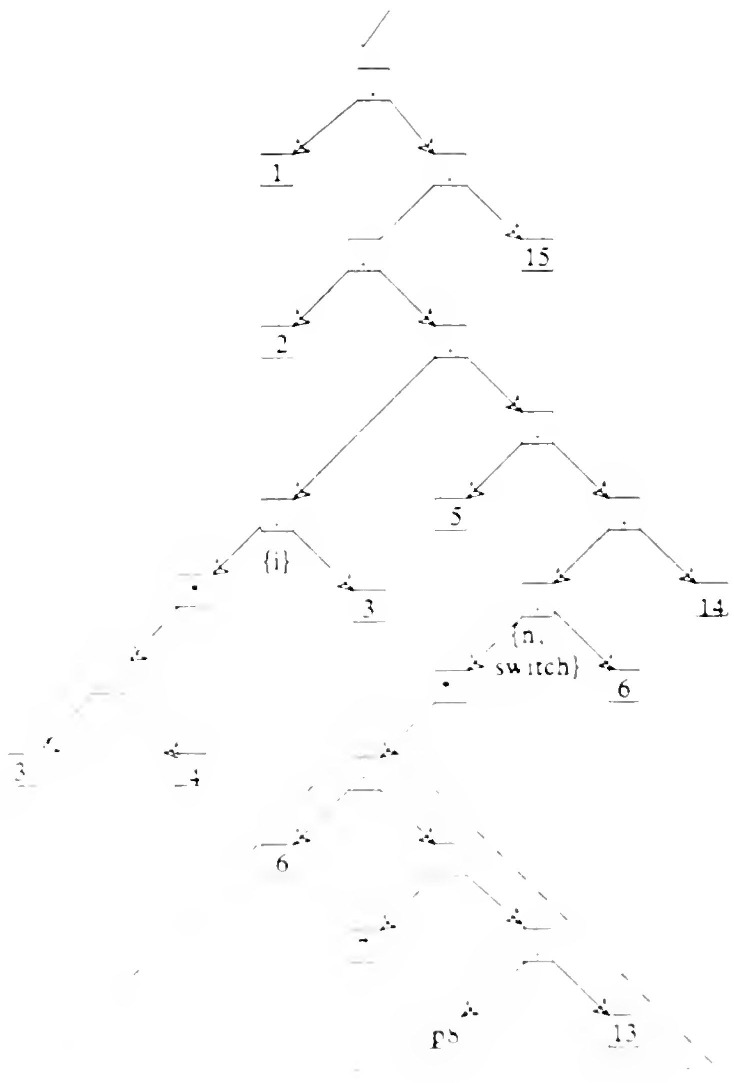
Apply RW2' to p3

(b)



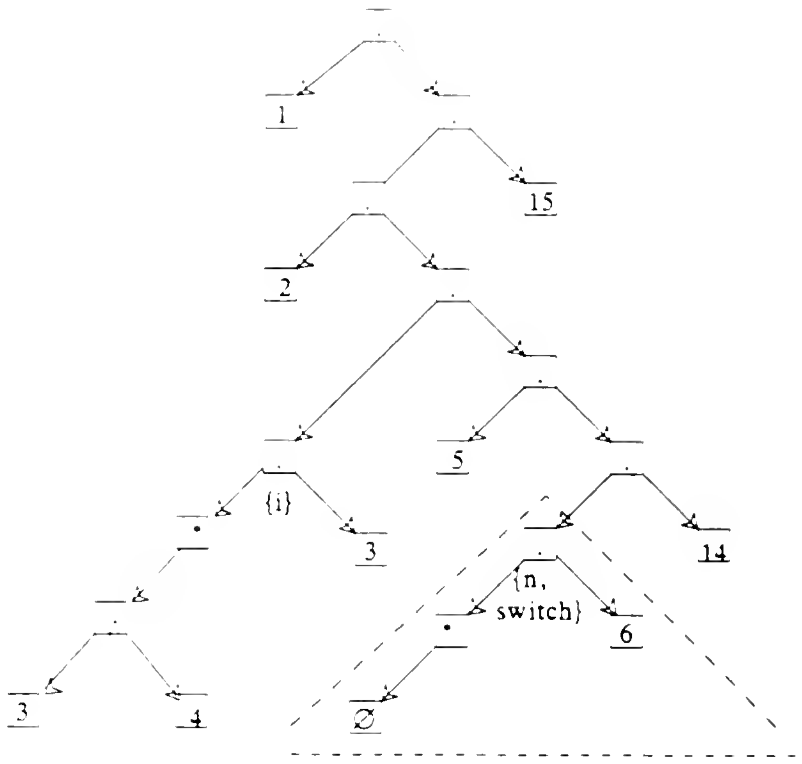
Apply RW2' to p6

(c)



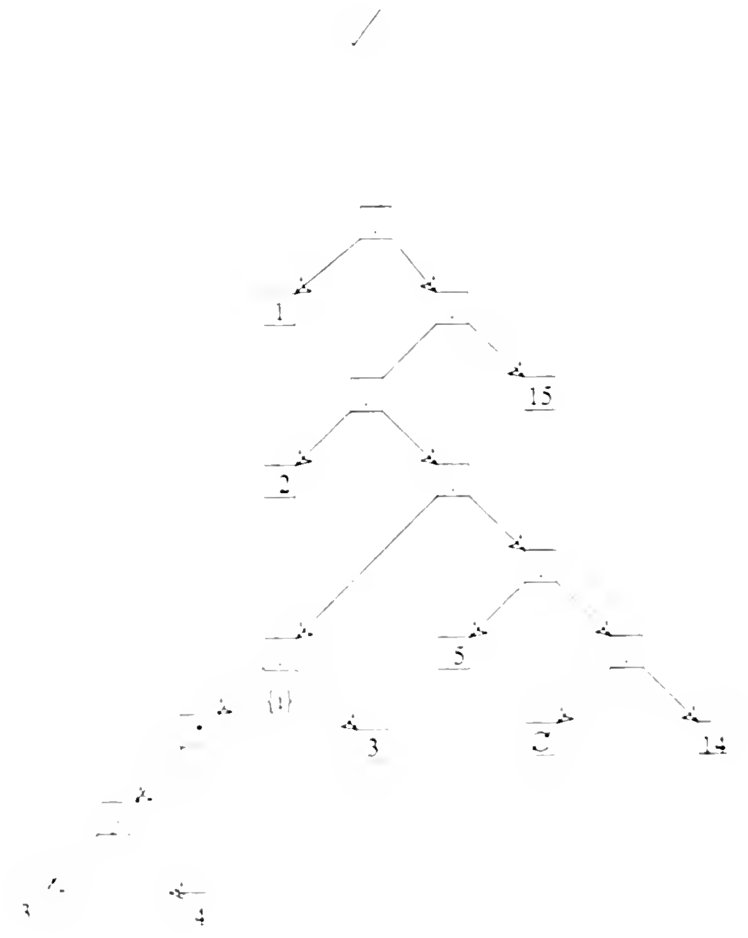
Apply RW2

(d)



Apply RW1

(e)



Simplify

(f)

Example 5.5.2

We next examine a somewhat more complicated, though admittedly contrived, example of the path expression method. Consider the following procedure, where "def(x)" and "use(x)" are abbreviations for some statements which have a definition or use, respectively, of the variable x. The flow graph for this procedure is shown in Figure 5.5.2.

```
procedure example5_5_2;
var x,y,z : integer;
function f(w:integer): boolean;
begin {f}
    ...
end; {f}
begin
        def(x);
        for y := 1 to 5 do
                for z := 1 to 5 do
                        if f(z) then
                        begin
                                def(x);
                                def(y)
                        end;
        use(x)
end;
```

Consider the definition-use association (2,11,x). The path expression method initializes ξ to $\mathbf{1 \cdot 2 \cdot p_3 \cdot 11 \cdot 12}$ and $\alpha^{d,u}$ to $\mathbf{2 \cdot p_3 \cdot 11}$. (See Fig 5.5.3a.)

We first concentrate on the loop symbol $\mathbf{p_3}$. The $T^{\mathbf{p_3}}$-reduction of T^{ξ} is ξ. β= $\mathbf{1 \cdot 2 \cdot p_3}^{top}$ and γ=$\mathbf{p_3}^{bot}\mathbf{\cdot 11 \cdot 12}$. E_1=PPC^{γ}≡$(y>5)$ and $PCOMP^{\beta}$⇒$(y=0)$. E_1 is β-unsatisfiable so the element {y} is added to MUST-REDEFINE($T^{\mathbf{p_3}}$).

Next, $T^{\mathbf{p_3}}$ is replaced by T^{δ} where δ=$\mathbf{(3 \cdot 4 \cdot p_5 \cdot 10)^* \cdot 3}$ is equivalent to $\mathbf{p_3}$. (See Fig 5.5.3b.) Since {y} is not contained in NEVER(δ) and there is no subexpression of δ in which x is always redefined, it is not possible to apply RW1 at this point. Control returns to the top of the while loop in PEM2 with ξ=$\mathbf{1 \cdot 2 \cdot (3 \cdot 4 \cdot p_5 \cdot 10)^* \cdot 3 \cdot 11 \cdot 12}$ and $\alpha^{d,u}$ to

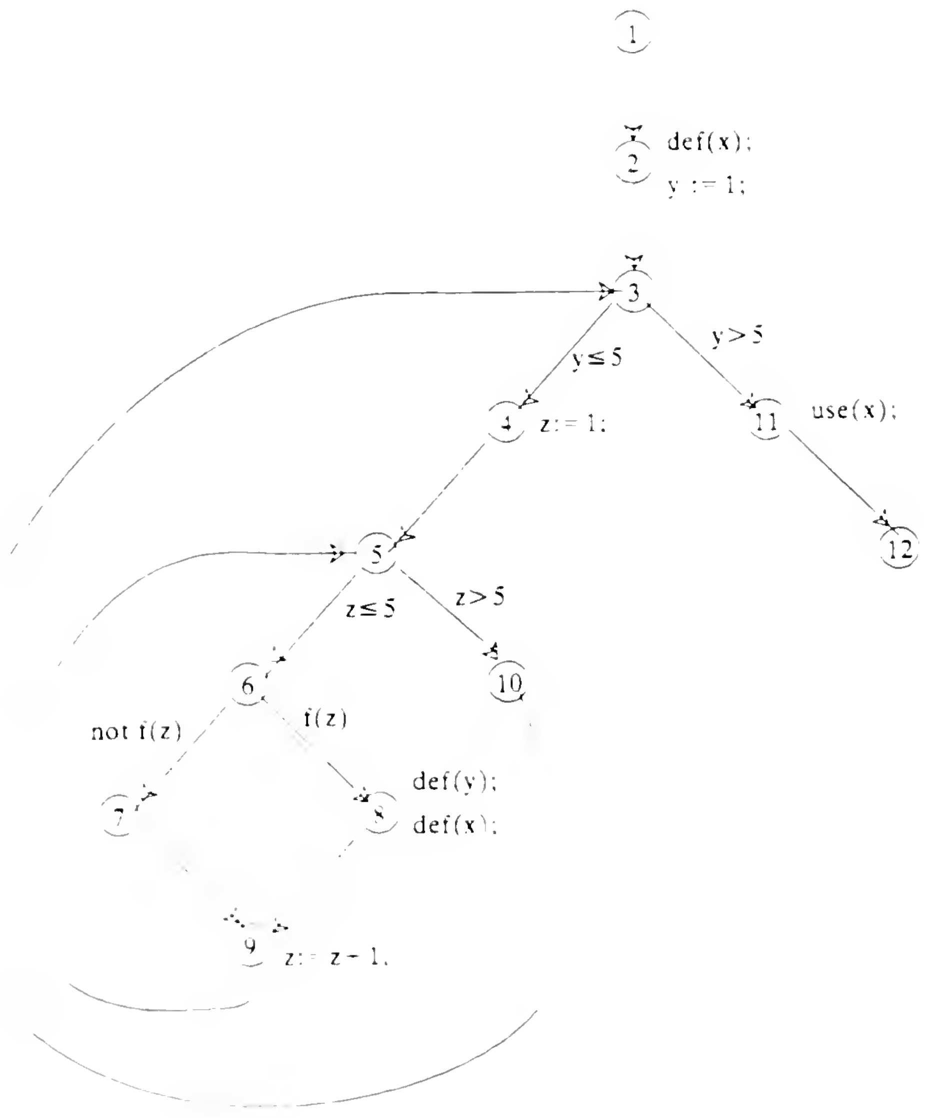
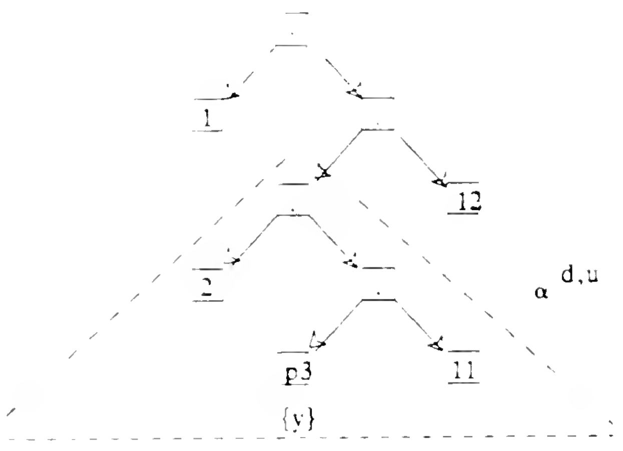

1
2
def(x);
y := 1;
3
y≤5
y>5
4
z:= 1;
11
use(x);
12
5
z≤5
z>5
6
10
not f(z)
f(z)
7
8
def(y);
def(x);
9
z:= z+1;

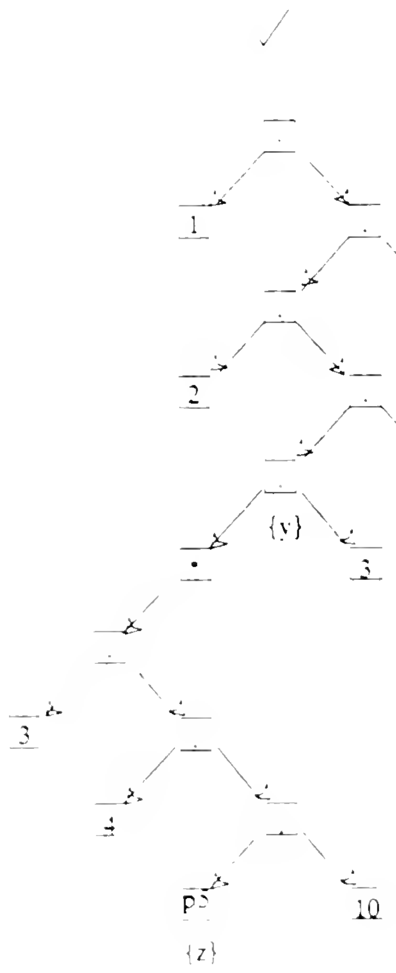
FIGURE 5.5.2

FIGURE 5.5.3

(a)

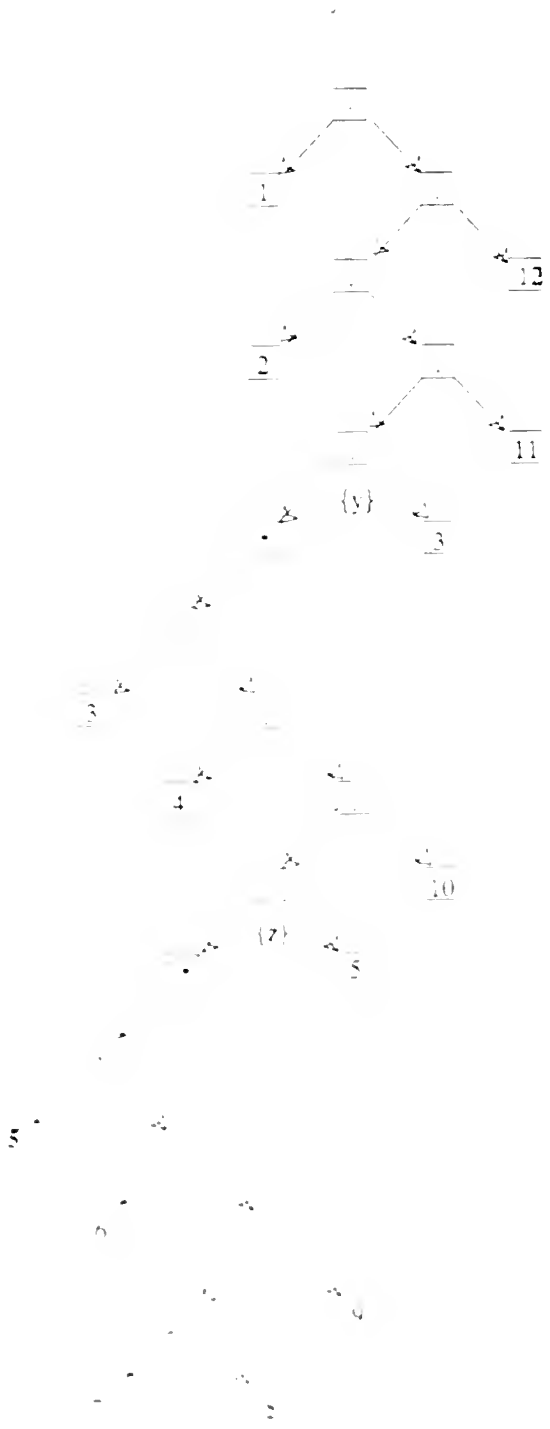


Apply RW2' to p3



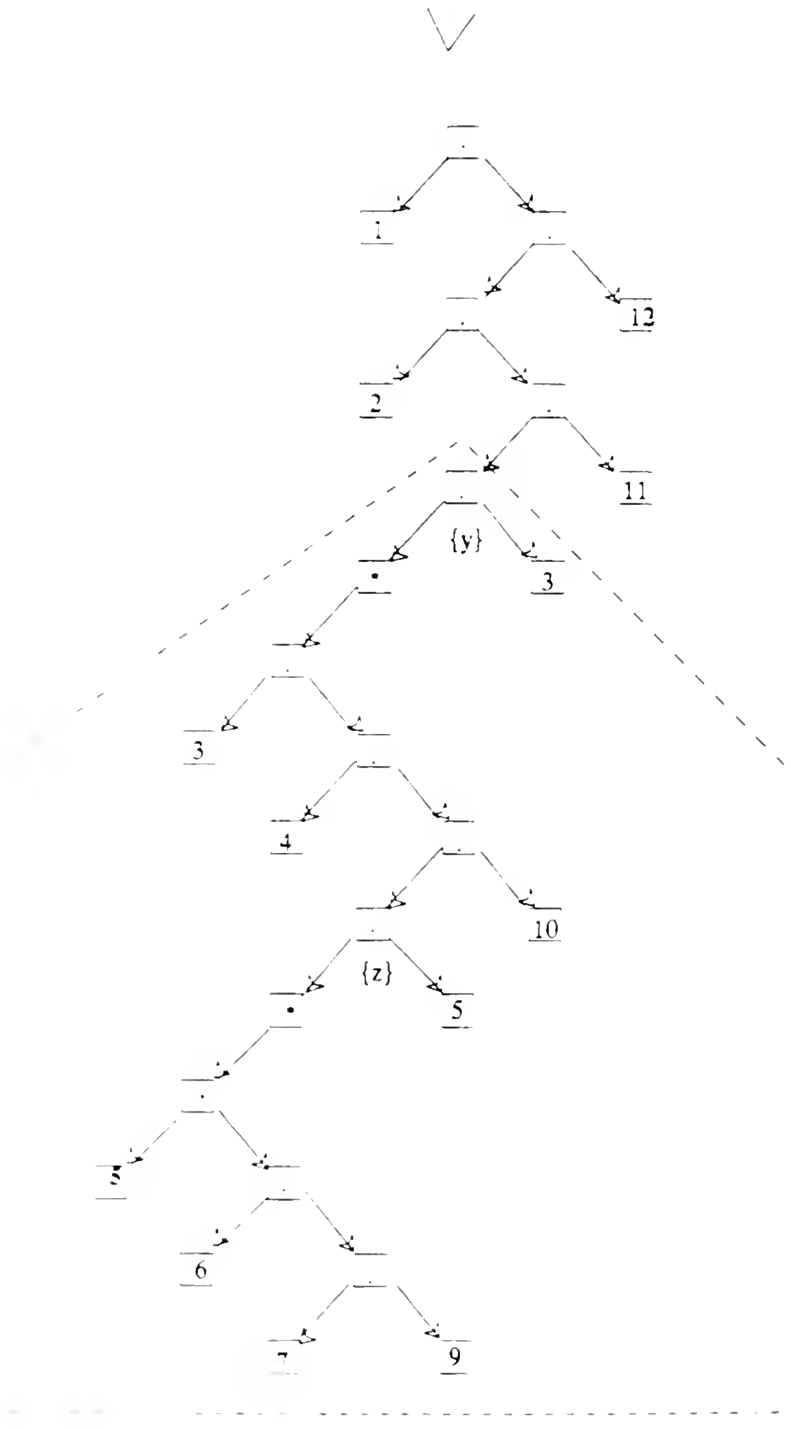
Apply RW2' to p5

(c)



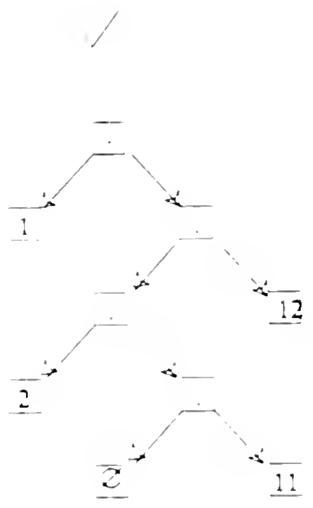
Apply RW1 to 8 and simplify

(d)



Apply RW1

(e)



Simplify

$\mathbf{2 \cdot (3 \cdot 4 \cdot p_5 \cdot 10)^* \cdot 3 \cdot 11}$.

We next concentrate on the loop symbol $\mathbf{p_5}$. The $T^{\mathbf{p_5}}$-reduction of T^{ξ} is $\mathbf{1 \cdot 2 \cdot 3 \cdot 4 \cdot p_5 \cdot 10 \cdot 3 \cdot 11 \cdot 12}$. $\beta = \mathbf{1 \cdot 2 \cdot 3 \cdot 4 \cdot p_5}^{top}$ and $\gamma = \mathbf{p_5}^{bot} \cdot \mathbf{10 \cdot 3 \cdot 11 \cdot 12}$. $E_1 = PPC^{\gamma} \equiv (z > 5)$ and $PCOMP^{\beta} \Rightarrow (z = 0)$. E_1 is β-unsatisfiable so the element $\{y\}$ is added to MUST-REDEFINE($T^{\mathbf{p_5}}$). Next, $T^{\mathbf{p_5}}$ is replaced by T^{δ} where $\delta = \mathbf{(6 \cdot (7 \cup 8) \cdot 9)^* \cdot 5}$ is equivalent to $\mathbf{p_5}$. (See Fig 5.5.3c.) Since x is always defined in node 8, RW1 can be applied. (See Fig 5.5.3d.) Now $\xi = \mathbf{1 \cdot 2 \cdot (3 \cdot 4 \cdot (6 \cdot 7 \cdot 8 \cdot 9)^* \cdot 5 \cdot 10)^* \cdot 3 \cdot 11 \cdot 12}$. Since $\{y\} \in$ MUST-REDEFINE($\mathbf{(3 \cdot 4 \cdot (6 \cdot 7 \cdot 8 \cdot 9)^* \cdot 5 \cdot 10)^* \cdot 3}$) and $\{y\} \subseteq$ NEVER($\mathbf{(3 \cdot 4 \cdot (6 \cdot 7 \cdot 8 \cdot 9)^* \cdot 5 \cdot 10)^* \cdot 3}$), this subexpression is RW1-safe. Applying RW1 gives $\xi = \mathbf{1 \cdot 2 \cdot \varnothing \cdot 11 \cdot 12}$ (See Fig 5.5.3e.) and simplifying gives $\xi = \varnothing$. So PEM2 exits from the while loop and returns "UNEXECUTABLE". Note that when $\{y\}$ was added to MUST-REDEFINE($T^{\mathbf{p_3}}$), $\{y\}$ was not contained in NEVER($\mathbf{p_3}$) so it was not possible to apply RW1 at the root R of this subtree. However, after applying RW2′ to $T^{\mathbf{p_3}}$ then to $T^{\mathbf{p_5}}$, and applying RW1 to $T^{\mathbf{8}}$ more details were known about the viable paths. At this point the subtree rooted at R became RW1-safe.

6. Enhancements and Limitations

In this section we will discuss limitations of the path expression method and propose some enhancements. These enhancements include the relaxation of some of the assumptions about the procedure being tested, as well as the use of more powerful path expression rewriting rules and symbolic evaluation techniques. We will also briefly discuss other uses of the path expression method. Throughout this section, when we refer to the "path expression method" we will mean the implementation of function PEM2 discussed in section 5.5, unless otherwise noted.

Since it is undecidable whether or not a particular definition-use association is executable, it is inevitable that the path expression method (even in its more general form) cannot identify all unexecutable associations. Rather than trying to formally classify those programs for which the path expression is guaranteed to be successful, we will examine a situation in which the path expression fails to identify unexecutable associations as such.

Example 5.6.1:

Consider the procedure *GetWord* in the following program:

```
program driver(input,output);
const MAX = 5;
type  String = array[1..MAX] of char;
var   a : String;
      i,length : integer;
procedure GetWord(var word : String; var n: integer);
{Reads a word and changes lower case letters into caps}
var   i      : 0..MAX;
      offset : integer;
begin
      writeln('Enter a word of length <= ',MAX:1);
      n := 0;
      while ((not eoln) and (n < MAX)) do
      begin
            n := n + 1;
            read(word[n])
      end;
      offset := ord('a') - ord('A');
      for i := 1 to n do
            if ('a' <= word[i]) and (word[i] <='z')
            then word[i] := chr( ord(word[i]) - offset)
end; {GetWord}

begin {driver}
      GetWord(a,length);
      for i := 1 to length do write(a[i]);
      writeln,
end.
```

The flow graph for this procedure is shown in Figure 5.6.1. We will now apply the path

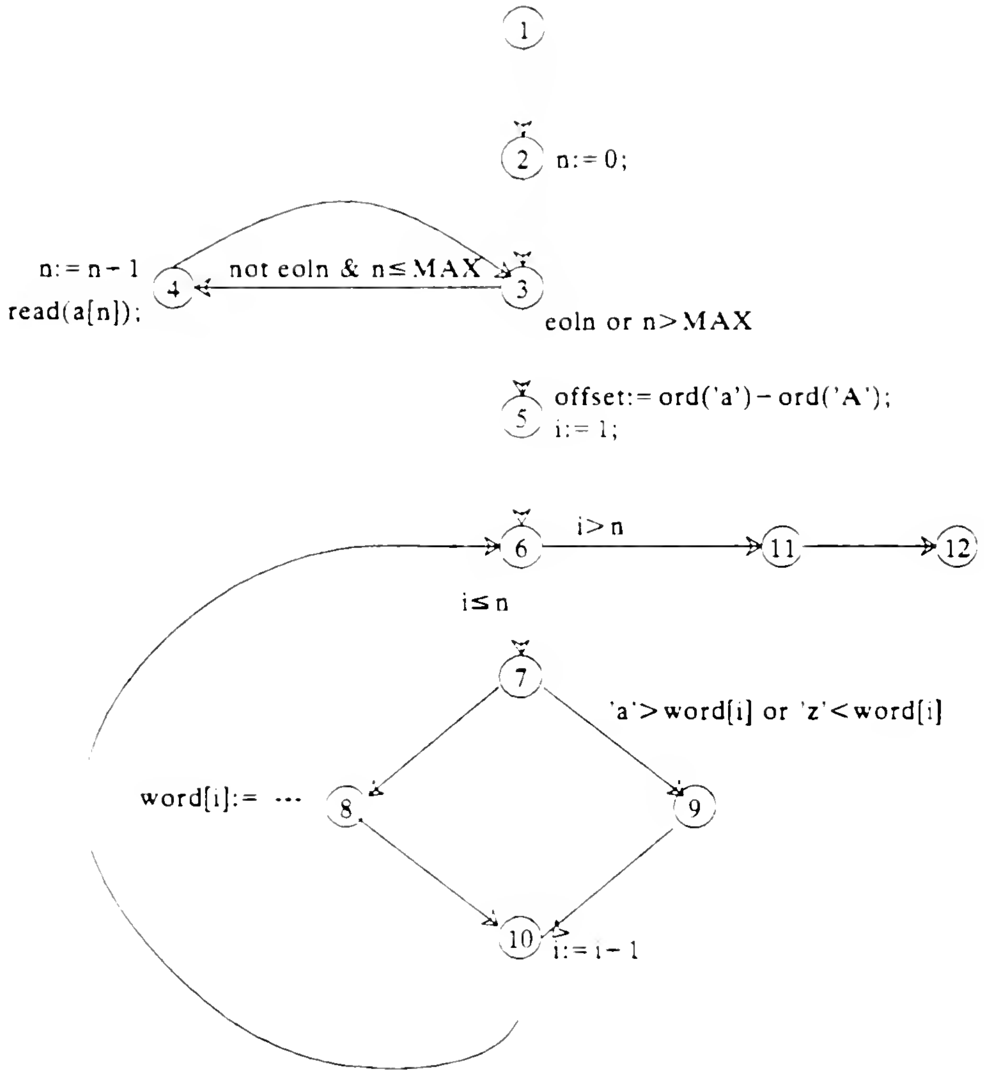

1
2
n:= 0;
n:= n+1
read(a[n]);
4
not eoln & n≤MAX
3
eoln or n>MAX
5
offset:= ord('a') − ord('A');
i:= 1;
6
i>n
11
12
i≤n
7
'a'>word[i] or 'z'<word[i]
word[i]:= ···
8
9
10
i:= i+1

Figure 5.6.1

expression method to the definition-use association (1,8,word). This association arises from the fact that *word* is defined in node 1 (since it is a **var** parameter) and used in node 8, and there is a definition clear path with respect to *word* from 2 to 8.

Initially,

$$\alpha^{d,u} = \mathbf{1}^{bot}\cdot\mathbf{2}\cdot\mathbf{p}_3\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{7}\cdot\mathbf{8}^{top}$$

and

$$\xi=\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{p}_3\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{7}\cdot\mathbf{8}\cdot\mathbf{10}\cdot\mathbf{p}_6\cdot\mathbf{11}\cdot\mathbf{12}.$$

We will first concentrate attention on the loop symbol $\mathbf{p}_3$. The $T^{\mathbf{p}_3}$-reduction of T^{ξ} is is ξ itself. $\beta=\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{p}_3{}^{top}$ and $\gamma=\mathbf{p}_3{}^{bot}\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{7}\cdot\mathbf{p}_8\cdot\mathbf{9}\cdot\mathbf{10}\cdot\mathbf{p}_6\cdot\mathbf{11}\cdot\mathbf{12}$.

To find a set of γ-necessary predicates we derive the partial path condition of γ:

$$E_1 \equiv PPC^{\gamma} \equiv (eoln \text{ or } n>5).$$

To check whether E_1 is β-unsatisfiable, we derive the partial path computation of β:

$$PCOMP^{\beta} \equiv (n=0).$$

Since the predicate $PCOMP^{\beta} \& E_1 \equiv (n=0) \& (eoln \text{ or } n>5) \equiv (eoln{=}\text{TRUE})$ is satisfiable, we cannot show E_1 to be β-unsatisfiable, and therefore cannot add any elements to the collection MUST-REDEFINE($T^{\mathbf{p}_3}$).

The path expression $\mathbf{p}_3$ is equivalent to $\delta=(\mathbf{3}\cdot\mathbf{4})^*\cdot\mathbf{3}$. Since *word* is defined in node 4, we apply RW1 to the subexpression **4** of δ. Simplifying gives $\delta'=\mathbf{3}$, and applying RW2′ gives $\alpha^{d,u}= \mathbf{1}^{bot}\cdot\mathbf{2}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{7}\cdot\mathbf{8}^{top}$, and $\xi=\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{p}_6\cdot\mathbf{7}\cdot\mathbf{8}\cdot\mathbf{10}\cdot\mathbf{p}_6\cdot\mathbf{11}\cdot\mathbf{12}$.

We next concentrate on the loop symbol $\mathbf{p}_6$. Let $T^{\mathbf{p}_6}$ denote the subtree corresponding to the occurrence of $\mathbf{p}_6$ in $\alpha^{d,u}$. The $T^{\mathbf{p}_6}$-reduction of T^{ξ} is is ξ itself. $\beta=\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{3}\cdot\mathbf{5}\cdot\mathbf{p}_6{}^{top}$ and $\gamma=\mathbf{p}_6{}^{bot}\mathbf{7}\cdot\mathbf{8}\cdot\mathbf{10}\cdot\mathbf{p}_6\cdot\mathbf{11}\cdot\mathbf{12}$. $PPC^{\gamma}\equiv(i\le n)\&(\text{'a'}\le word[i])\&(word[i]\le\text{'z'})$. Let $E_1=(i\le n)$. $PCOMP^{\beta}\Rightarrow(i=1)\&(n=0)$ so E_1 is β-unsatisfiable. So the element $\{i,n\}$ is placed in MUST-REDEFINE($T^{\mathbf{p}_6}$).

RW2′ is applied, replacing $\mathbf{p}_6$ by the equivalent path expression $\mathbf{(6{\cdot}7{\cdot}(8 \cup 9){\cdot}10)^*{\cdot}6}$ Since *word* is defined in node 8 (or since node 8 is the target node) RW1 is applied to node 8. At this point, $\alpha^{d,u}= \mathbf{1}^{bot}\mathbf{{\cdot}2{\cdot}3{\cdot}5{\cdot}(6{\cdot}7{\cdot}9{\cdot}10)^*{\cdot}6{\cdot}7{\cdot}8}^{top}$, and $\xi=\mathbf{1{\cdot}2{\cdot}3{\cdot}5{\cdot}(6{\cdot}7{\cdot}9{\cdot}10)^*{\cdot}6{\cdot}7{\cdot}8{\cdot}10{\cdot}p_6{\cdot}11{\cdot}12}$. Since {i,n} is not a subset of NEVER($\mathbf{(6{\cdot}7{\cdot}9{\cdot}10)^*{\cdot}6}$), no further re-writing can be done at this point.

Since $\alpha^{d,u}$ has no remaining loop symbols, we exit from the while loop and compute PPC^{ξ}. Since $PPC^{\xi}\equiv((n=0)\&eoln)$ is not equivalent to FALSE, the algorithm returns "POSSIBLY-EXECUTABLE".

However, the association is in fact unexecutable. This can be seen by observing that the value of *i increases* in the loop headed by node 6. Thus, since the predicate $(i=1)\&(n=0)$ holds before entering the loop and the value of n does not change in the loop, the exit condition, $(i>n)$, can never hold. So none of the candidates are viable.

Intuitively, the reason that the path expression method fails to conclude that this association is unexecutable is because it takes into account the fact that *i* is redefined in the loop, but fails to take into account *the way in which* it is redefined.

There are several ways to enhance the path expression method so that unexecutable associations like this one would be detected. One method would be to augment the procedure with user-supplied (or programmer-supplied) assertions and allow the partial symbolic evaluator to use the assertions in the obvious way. If the assertion "$i>1$" were added on branch (6,7) then partial symbolic evaluation of

$$\xi=\mathbf{1{\cdot}2{\cdot}3{\cdot}5{\cdot}(6{\cdot}7{\cdot}9{\cdot}10)^*{\cdot}6{\cdot}7{\cdot}8{\cdot}10{\cdot}p_6{\cdot}11{\cdot}12}$$

would yield $PPC^{\xi}\Rightarrow(n=0)\&\ eoln\ \&(i'>1)\&(i'<n)\equiv$FALSE where i' is the symbolic value of i after executing the sub-path expression $\mathbf{1{\cdot}2{\cdot}3{\cdot}5{\cdot}(6{\cdot}7{\cdot}9{\cdot}10)^*}$. Thus the algorithm would return "UNEXECUTABLE".

Another way to enhance the path expression method would be to use a more powerful symbolic evaluation technique than partial symbolic evaluation. While global symbolic evaluation in its full generality is not feasible, some limited form in which recurrence relations are derived and solved for *some* but not *all* variables could be practical. For example, the recurrence relation for variable i in the loop headed by node 6 is simple:

$i^0=1$

$i^k=i^{k-1}+1$ for k>0

where i^k denotes the value of i after k iterations of the loop. Solving the recurrence relation yields i^k=k+1 for k≥0 which implies that the value of i is greater than or equal to 1 when branch (6,7) is executed. An enhanced version of the path expression method which employed symbolic evaluation techniques powerful enough to infer this fact would return "UNEXECUTABLE" for this association.

Another approach to enhancing the path expression method is to include more powerful rewrite rules. Restricting the rewrite rule RW2 to RW2′ was a severe restriction, aimed at ensuring the efficiency of the algorithm. If somewhat less stringent restrictions were placed on RW2, a more powerful algorithm would result. Consider for example the rewrite rule

RW2″:

Replace a subtree T^{α} of T^{ξ} by T^{β} where $\alpha=(\gamma\cdot\delta)^{*}\cdot\gamma$ and $\beta=\gamma\cdot(\delta\cdot\gamma)^{*}$.

If the path expression method were enhanced to allow application of RW2″ as well as RW1, and RW2′ then $\xi=1\cdot2\cdot3\cdot5\cdot(6\cdot7\cdot9\cdot10)^{*}\cdot6\cdot7\cdot8\cdot10\cdot p_6\cdot11\cdot12$ could be replaced by $\xi'=1\cdot2\cdot3\cdot5\cdot6\cdot7\cdot(9\cdot10\cdot6\cdot7)^{*}\cdot8\cdot10\cdot p_6\cdot11\cdot12$. Since $PPC^{\xi'}\equiv$FALSE, the path expression method would return "UNEXECUTABLE".

More research is needed to determine which rewrite rules and which more powerful symbolic evaluation techniques could be added to the path expression method without unduly compromising efficiency.

We next discuss the prospects for relaxing some of the assumptions, made in section 5.1, about the procedure P being tested. We will discuss the way various strategies for relaxing assumptions affect the power of the path expression heuristic.

The first assumption about P was that no aliasing occurs in P. There are several ways in which aliasing can arise in a procedure: If *x* and *y* are **var** formal parameters in P and P is called with the same actual parameter passed to *x* and *y* then *x* and *y* are aliases. Similarly, if *x* is a global variable which is also passed to the **var** parameter *y* then *x* and *y* are aliases. If *p* and *q* are pointer variables with the same value then *p*ˆ and *q*ˆ are aliases. If *a* is an array and i and j have the same value then *a* [*i*] and *a* [*j*] are aliases.

In order for the path expression method to function correctly it is necessary that all aliasing be known. If *x* and *y* are aliased then any assignment to *x* is also an assignment to *y*. This must be taken into account both in the symbolic evaluation and in the checks for whether a variable is in ALWAYS(α), SOMETIMES(α), or NEVER(α) for a path expression α. Two standard approaches to insuring that all aliasing is known are to restrict the kind of aliasing which can occur, and to allow any kind of aliasing but make conservative assumptions about the values of variables.

The later approach entails considering each definition of a variable *x* to also be a definition of variable *y* if *x* and *y* can potentially be aliased. This is the approach we have taken to handling arrays. One could apply this approach to other kinds of aliasing as follows: Replace NEVER(α) by *NEVER'*(α) and SOMETIMES(α) by *SOMETIMES'*(α) where *NEVER'*(α)={*x*| neither *x* nor any variable which is potentially aliased to *x* is defined on any element of PATHS(α)}, and where *SOMETIMES'*(α)= complement of *NEVER'*(α). In partial symbolic evaluation of a statement in which a variable x is defined, it would be necessary to also note that any variable which is potentially aliased to x could also change value. For example, if y is a potential alias of x, then after symbolically executing the statement "$x := z$", the new symbolic value, Y of y is given by Y=Z or Y=Y', where Y' is the old symbolic value of y and Z is the symbolic value of z. If there are a lot of variables which are potential aliases, this can become quite unwieldy.

A simpler approach to partial symbolic evaluation in the presence of aliasing is to assign a unique new symbol to each potential alias of x whenever an assignment to x is made. This is computationally simple, but could result in too much loss of information. Most likely, the best method of relaxing the "no aliasing" assumption would involve a combination of restricting the type of aliasing which could occur and making conservative assumptions about the ways in which values of variables can change. Perhaps heuristic approaches to determining whether variables which are *potential* aliases are *actually* aliased would be fruitful.

The second assumption about P (no hidden side-effects and no functions with **var** parameters) was also motivated by concern for ensuring the accuracy of the data flow information collected and the partial symbolic evaluation. Approaches to relaxing this assumption are similar to those to relaxing the first assumption. By performing interprocedural data flow analysis [MUC81,BUR86], it is possible to "uncover" hidden side effects. In those cases where it is not possible to tell for certain whether a called procedure or function has side effects it is necessary to make conservative assumptions.

The third assumption is that functions have no **var** parameters. This assumption is a convenience which simplifies the symbolic evaluation of function calls. It can easily be eliminated.

The fourth assumption, that P has no **repeat-until** statements, was included as a technical convenience in order to insure that all loops in P's flow graph have exits from the top. One way to relax this assumption would be to have different kinds of loop symbols for **repeat-until** loops and for **while** loops and to expand the set of rewrite rules accordingly. It also seems likely that the path expression method could be generalized to handle arbitrary reducible flow graphs. The concept of loop symbols representing the set of paths through a loop would be replaced by "interval symbols" representing the paths through an interval. For arbitrary flow graphs, some variant of PE1 could probably be developed. Such an algorithm would use regular expressions without loop symbols, and would initialize the path expression ξ using the algorithm in [TAR81B]. More investigation of these ideas is needed.

Up to this point we have focussed on the use of the path expression method for attempting to determine that a given definition-use association is unexecutable. This has primarily been motivated by the fact that if we show that all of the definition-use associations required by a data flow testing criterion C but not covered by a test set T are unexecutable then we have shown that T is C*-adequate. We will now briefly discuss two other applications of the path expression method.

The path expression method can also be used as an aid for test data generation. Suppose (d,u,v) is an *executable* definition use association. If we apply the path expression method to (d,u,v) it will return "Possibly Executable". The final value of ξ will be a path expression representing the final set of candidates. This may be a proper subset of the initial set of candidates. By partially symbolically executing the path expression ξ we obtain the ξ-necessary predicate PPC^{ξ}. Since any test element which covers (d,u,v) must satisfy PPC^{ξ}, we need only consider such elements as potential test cases. Note however, that while PPC^{ξ} is a necessary condition for a test case to cover (d,u,v) it is not sufficient.

The path expression method may also be useful in compiler optimization. Many of the code transformations performed during optimization involve finding the definitions of a variable which *reach* a particular point p, that is, definitions d of variable v such that there is a definition clear path with respect to v from d to p. Compilers generally make the conservative assumption that all definitions which reach a point are relevant. However, if the only definition clear paths with respect to v from d to p are unexecutable, then the value which is assigned to v at d cannot effect the value of v at p. In our notation, if the definition-use association (d,p,v) is unexecutable, then the optimizing compiler can safely remove d from the set of definitions of v which reach p.

Consider, for example, constant propagation. Suppose that definitions $d_1,...,d_k$ of variable v all reach point p. Suppose that at $d_2,...,d_k$, the value assigned to v is zero, but at d_1 it is non-zero. If the definition-use association (d_1,p,v) can be shown by the path expression method to be unexecutable, then whenever control reaches point p, the value of v is guaranteed to be zero. Thus any use of v in the statement at point p can be replaced by the constant zero. Similarly, the path expression method could be used to increase the accuracy of copy propagation, live-variable analysis, detection of induction variables, detection of parallelism, etc.

More work is needed to determine when, whether, and to what extent the cost of using the path expression method to eliminate "unexecutable" reaching definitions is justified by the improvement in the compiled code.

CHAPTER 6:

Conclusion

1. Summary

In this thesis we have investigated two families of test data adequacy criteria, the data flow testing criteria and the feasible data flow testing criteria. We have used the inclusion relation to compare these criteria to one another and to statement, branch and path testing; have explored the extent to which each criterion satisfies Weyuker's axioms for a good adequacy criterion; and have examined prospects for building testing tools based on the criteria. We have described the design and implementation of a tool, ASSET, which is based on data flow testing, and have described a heuristic, the path expression method, which, in conjunction with ASSET partially automates feasible data flow testing.

The data flow testing criteria are based purely on the syntax of the program being tested. A definition use association is either

1) a triple (d,u,v), where d is a node having a definition of variable v, u is a node having a use of v, and there is a definition-clear path with respect to v from d to u, or

2) a du-path with respect to v.

The criteria require the test data to exercise certain subsets of the set of definition-use associations. These criteria were originally defined by Rapps and Weyuker to apply to programs written in a simple language. We have extended the definitions to apply to subprograms written in a large subset of Pascal. This has been done in such a way as to preserve the inclusion relationships among the criteria.

The data flow testing criteria fail to satisfy several of Weyuker's axioms for a "good" adequacy criterion. In particular, they fail to satisfy the applicability axiom, which requires that an adequate test set exist for each subprogram. Furthermore, it is undecidable whether an adequate test set exists for a given subprogram. This is not surprising, since the data flow testing criteria are structured testing criteria, and all of the structured testing criteria have this weakness. For the data flow testing criteria, even quite "ordinary" subprograms may fail to be adequately testable.

The feasible data flow testing criteria are derived from the data flow testing criteria by eliminating from consideration those definition-use associations which can never be exercised. All but one of the feasible data flow testing criteria satisfy all of Weyuker's axioms, including applicability property. However, given test set T, an FDF testing criterion C, and a subprogram P, it is undecidable whether T is C-adequate for P.

We have shown that the inclusion relations among the feasible data flow testing criteria differ from those among the data flow testing criteria. While all-du-paths $\Rightarrow$ all-uses, (all-du-paths)* is incomparable to all of the other feasible data flow testing criteria. Also, while all-p-uses $\Rightarrow$ all-edges, (all-p-uses)* and (all-edges)* are incomparable. This means that the feasible data flow testing criteria do not "bridge the gap" between (the feasible analogs of) branch testing and path testing. However, if for class of subprograms satisfying the quite reasonable NFUP property, defined in chapter 4, (all-p-uses)* $\Rightarrow$ (all-edges)*.

The fact that the relationship among the feasible data flow testing criteria differs from the relationship among the data flow testing criteria has implications for studies of the formal properties of other testing criteria. Several criteria which fail to satisfy the applicability property (including all of the structured testing criteria) have been defined and some of their formal properties have been investigated. The criteria which are actually used in practice, however, are in the spirit of the feasible data flow testing criteria. It is these criteria, which take account of semantic information which should be studied, not their purely-syntactic counterparts.

For example, consider a practitioner who is trying to branch test a subprogram P with an unexecutable branch b. At some point, observing that none of the test cases have caused b to be executed, she will examine P, determine that b is unexecutable, and eliminate it from consideration. Thus, the criterion being used is actually (all-edges)*, not all-edges.

The theorems appearing in the literature which compare various structured testing criteria should be re-examined in this light. That is, for each of these criteria C, a criterion C* should be defined, which agrees with C for programs having no unexecutable paths, but which satisfies the applicability property. The relationship among the starred criteria should be then examined. If criterion C_1 does not include criterion C_2 then C_1* will not include criterion C_2*. However, if $C_1 \Rightarrow C_2$ it will not necessarily be the case that $C_1{}^* \Rightarrow C_2{}^*$.

In order to determine whether a test set T satisfies a feasible data flow testing criterion C, it is necessary to examine all of the definition-use associations required by C but not covered by T to see whether any of them are executable. This may be a tedious and error prone task. With that in mind, we have developed a heuristic method, the path expression method, which attempts to determine whether a given definition-c-use association or definition-p-use association is executable.

Given a definition-use association (d,u,v), a viable candidate is an executable path from subprogram entry to subprogram exit, which has a definition clear subpath with respect to v from d to u. The path expression method uses a regular expression ξ to represent a superset of the set of viable candidates. A combination of symbolic evaluation and data flow information is used to weed out paths which can be shown not to be viable candidates, and ξ is rewritten accordingly. If, when the heuristic terminates, ξ represents the empty set of paths, then the association (d,u,v) is guaranteed to be unexecutable.

The path expression method uses a new symbolic evaluation technique called partial symbolic evaluation of path expressions. Roughly speaking, partial symbolic evaluation of a path expression α which represents a single path π is equivalent to symbolic execution of π, while partial symbolic evaluation of a path expression α which represents more than one path assigns a new symbolic value to those variables which are potentially redefined on any one of the paths. The result of partially symbolically evaluating path expression α is an expression which describes in part the effect of executing any one of the paths represented by α.

The path expression method and partial symbolic evaluation of path expressions were developed in order to help eliminate from consideration unexecutable definition-use associations. They can also be used as an aid to test data generation. If the path expression method returns the answer, "association may be executable", then ξ represents a non-empty set of paths, which includes all of the viable candidates. Symbolic evaluation of ξ gives a predicate which must be satisfied by any test case which covers the given definition-use association.

We believe that the path expression method will be able to identify a significant number of the unexecutable definition-use associations in "typical" subprograms. In conjunction with ASSET this will form a usable tool for feasible data flow testing.

Typically, the user would invoke the path expression heuristic when, after a "reasonable" amount of testing, some definition-use associations remained unexecuted. If the heuristic determined that the association was unexecutable, the user could safely eliminate it from consideration. Otherwise the user would be left to determine by hand whether that association was in fact unexecutable, or to find test data to cover the association. The final path expression ξ produced by the heuristic would provide some helpful information to the user. Preliminary experiments indicate that this is a reasonable scenario, but more experimentation is needed.

2. Future Work

There are many avenues for future research. Several questions relating to the enhancement of the path expression method and of symbolic evaluation of path expressions are suggested in Chapter 6, section 5. Some other topics for future work include

• Empirical and theoretical studies of the effectiveness of feasible data flow testing:

Such studies would be aimed at measuring the error detecting ability of the criteria. In order to perform empirical studies, a large sample of "real" programs, preferably with known errors, would be needed.

Theoretical studies would have as their goal the classification of those errors which are guaranteed to be detected by the various criteria. Theoretical and empirical studies of the amount of test data required by the criteria would also be useful. A related issue is the classification of program features which make feasible data flow testing more or less effective.

• Empirical and theoretical studies of effectiveness of heuristic:

Empirical studies would be aimed at determining the percentage of unexecutable associations which are identified as such by the heuristic. We are currently conducting a pilot study in which we are examining the subprograms in Kernighan and Plauger's *Software Tools in Pascal* [KER 81].

Theoretical studies would attempt to classify those unexecutable associations which are guaranteed to be identified. A related issue is the classification of program features which make identifying unexecutable associations harder or easier.

• Investigate the effect of using more sophisticated data flow analysis:

The data flow analysis used to define the DF and FDF criteria was quite unsophisticated. It seems reasonable to expect that using more sophisticated data flow analysis techniques would lead to criteria which demanded "better" test data. For example, using interprocedural data flow analysis, one could define criteria which apply to larger program units than procedures and functions. It might also be possible to define criteria which treat array elements as individual variables, using heuristics to attempt to disambiguate array references. One could then explore the trade-off between difficulty of applying the criteria to a program and test set, versus error-detecting ability of the test sets demanded by the criteria.

• Extend data flow testing and feasible data flow testing to apply to programs written in other languages:

Depending on the language in question, this could involve handling minor issues, such as static variables and definitions within expressions, and/or major issues such as those arising in languages which explicitly support parallelism.

BIBLIOGRAPHY

[AHO86] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Mass., 1986.

[BUR86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *SIGPLAN '86 Symposium on Compiler Construction*, ACM.

[CHE] T.E. Cheatham, Jr., G.H. Holloway, J.A. Townley, "Symbolic Evaluation and the Analysis of Programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, July 1979.

[CLA81] L.A. Clarke, D.J. Richardson, "Symbolic Evaluation Methods -- Implications and Applications," in *Computer Program Testing*, ed. by Chandrasekaran and Radicchi, North-Holland Publishing Co., 1981.

[CLA85] L.A. Clarke, A. Podgurski, D.J. Richardson and S.J. Zeil, "A Comparison of Data Flow Path Selection Criteria", *8th IEEE International Conference on Software Engineering*, August, 1985, London, pp. 244-251.

[DAV73] M.D. Davis, "Hilbert's Tenth Problem is Unsolvable," American Mathematical Monthly, 80, March 1973.

[DAV83] M.D. Davis, E.J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, New York, 1983.

[FOS76] L.D. Fosdick,L.J Osterweil, "Data Flow Analysis in Software Reliability," *ACM Computing Surveys*, 8,3,1976.

[FRA85a] P.G. Frankl, S.N. Weiss and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.

[FRA85b] P.G. Frankl and E.J. Weyuker, "A Data Flow Testing Tool," *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.

[FRA87a] P.G. Frankl, "ASSET User Manual," Computer Science Department Technical Report #318, Courant Institute of Mathematical Sciences, New York University, New York, NY, Sept 1987.

[GIR85] M.R. Girgis and M.R. Woodward, "An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis," *8th IEEE International Conference on Software Engineering*, August, 1985, London, pp. 313-319.

[GOO75] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, SE-1,2, June 1975.

[GOU83] J.S. Gourlay, "A Mathematical Framework for the Investigation of Testing", *IEEE Transactions on Software Engineering*, SE-9, 6, Nov. 1983.

[HAR77] W.H. Harrison, "Compiler Analysis of the Value Ranges for Variables", *IEEE Trans. Software Eng.*, Vol. SE-3, No.3, 1977.

[HEC77] M. S. Hecht, *Flow Analysis of Computer Programs*, North Holland, 1977.

[HER76] P.M. Herman, "A Data Flow Analysis Approach to Program Testing", *The Australian Computer Journal*, Vol. 8, No. 3, November 1976.

[HOW76] W.E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, 1976, pp. 208-215.

[HOW78] W.E. Howden, "A Survey of Dynamic Analysis Methods" in *Tutorial: Software Testing and Validation Techniques*, eds. E. Miller and W.E. Howden, IEEE Computer Society, 1978.

[HOW78b] W.E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," *Software Practice and Experience*, 8, 1978.

[HUA75] J.C. Huang, " An Approach to Program Testing," *ACM Computing Surveys*, 7(3), Sept. 1975, pp. 113-128.

[JEN85] K. Jensen, N. Wirth, *Pascal User Manual and Report, Revised for the ISO Pascal Standard,* Springer, New York, 1985.

[KEM85] R.A. Kemmerer, S.T. Eckmann, "UNISEX: a UNIx-based Symbolic EXecutor for Pascal," *Software Practice and Experience*, Vol. 15(5), May 1985.

[KER81] Kernighan, Plauger, *Software Tools in Pascal*, Addison Wesley, Reading, Mass., 1981.

[KNU77] D.E. Knuth, J.H. Morris, V.R. Pratt, "Fast Pattern Matching in Strings," SIAM Journal on Computing, 6, 2, June, 1977.

[KIN76] J.C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, 19, 7, July 1976.

[KOR85] B. Korel and J. Laski, "A tool for Data Flow Oriented Program Testing", *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.

[LAS83] J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Trans. Software Eng.*, Vol. SE-9, No.3, May 1983, pp. 347-354.

[MIL74] E. F. Miller, Jr. M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation," *Dig. Compcon*, Spring 1974, pp.161-164.

[MUC81] Muchnick, Jones, *Program Flow Analysis, Theory and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[NTA84] S. Ntafos, "On Required Element Testing," *IEEE Trans. Software Eng.*, Vol. SE-10, No.6, Nov. 1984, pp. 795-803.

[OST76] L.J. Osterweil and L. D. Fosdick, "DAVE -- A Validation Error Detection and Documentation System for Fortran Programs," *Software Practice and Experience*, Oct-Dec 1976, pp. 473-486.

[OST81] L.J. Osterweil, "Using Data Flow Tools in Software Engineering," in *Program Flow Analysis: Theory and Applications*, Muchnick, ed., Prentice Hall 1981.

[RAP82] S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Program Test Data Selection," *Proceedings Sixth International Conference on Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 272-278.

[RAP85] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE-11, No.4, April 1985, pp. 367-375.

[SCH73] M. Schaeffer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[TAR81A] R.E. Tarjan, "A Unified Approach to Path Problems," *Journal of the Association for Comoputing Machinery*, Vol. 28, No. 3, July 1981.

[TAR81B] R.E. Tarjan, "Fast Algorithms for Solving Path Problems," *Journal of the Association for Comoputing Machinery*, Vol. 28, No. 3, July 1981.

[WEG83] M. Wegman, "Summarizing Graphs by Regular Expressions," *Proceedings of the Tenth SIGACT POPL*, Austin, Tex., 1983.

[WEY79] E.J. Weyuker, "The Applicability of Program Schema Results to Programs," *International Journal of Computer and Information Sciences*, 8.5, Oct. 1979

[WEY82] E.J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, 25,4, 1982

[WEY84] E.J. Weyuker, "The Complexity of Data Flow Criteria for Test Data Selection," Information Processing Letters, Vol. 19, No. 2, August 1984, pp. 103-109.

[WEY86] E.J. Weyuker, "Axiomatizing Software Test Data Adequacy," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 12, Dec. 1986.

[WOO80] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with Path Analysis and Testing of Programs," *IEEE Trans. Software Eng.*, Vol. SE-6, May 1980, pp. 278-286.

APPENDIX I: ASSET USER MANUAL Preliminary Version June 29, 1987

Phyllis Frankl

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

1. INTRODUCTION

ASSET (A System to Select and Evaluate Tests) is an interactive software testing tool based on data flow testing [1,2,3,4]. It takes as input a Pascal program, a set of test data, and one of the data flow testing criteria. It runs the program on the test data and produces a list of those definition-use associations required by the given criterion but not covered by the test data.

The user then examines the output to determine whether the program has behaved according to its specification. If the program computed the wrong output, the user reports that the program has a bug. If the program has behaved correctly on all of the test cases, and if the criterion has not been satisfied, the user selects additional test data and continues the testing process. If the program has behaved correctly and the criterion has been satisfied the user can either release the program, certifying that it has been adequately tested according to the criterion, or can select another criterion and continue.

ASSET is used for testing an *individual subprogram* (procedure, function, or main program) which we'll call the "subject procedure". We will refer to the program of which the subject procedure is part as the "subject program". The data flow analysis which it performs is *intra-procedural* analysis of the subject procedure.

This manual assumes that the reader has some familiarity with the theory of data flow testing and with the UNIX operating system. To acquire some familiarity with data flow testing, see the references in section 7. Section 2 of this manual defines the restrictions on programs accepted by ASSET. Section 3 describes the most important ASSET commands. Section 4 describes the rest of the ASSET commands. Section 5 describes how to use ASSET on a separately compiled program. Section 6 gives an example of an ASSET session. Section 7 gives references to relevant papers.

Acknowledgements:

The graphics facility was written by Stewart Weiss, who also helped implement an earlier version of ASSET. The user-interface was translated from csh to C and enhanced by Ernie Campbell

Disclaimer:

ASSET is still a "work in progress". We do not vouch for its reliability nor are we responsible for maintaining it. However, if you find any bugs, or have other suggestions as to how the tool could be improved we would appreciate hearing from you. Please send e-mail to frankl@csd2.nyu.edu *and* to weyuker@csd2.nyu.edu.

2. THE LANGUAGE ACCEPTED BY ASSET

ASSET accepts *syntactically correct* programs written in the subset of (level 0) ISO Pascal consisting of programs which **do not** include any of the following constructs:

1. **goto** statements and labels.
2. functions with **var** parameters.
3. variant records
4. **with** statements
5. procedural or functional parameters
6. **forward** declarations
7. conformant arrays

The following identifiers are used in the modified version of the subject program and therefore should not be used in the subject program:

FW
traversed

The following identifiers are used as file names by ASSET and therefore should not be used as file names in the subject program:

IncFiles, StringTabFile, SymTabFile, anomalies, a.out, comp.msgs, copy, copy.exec, copy.p, copy2, copy2.p, defuse, display, globdecs, graph, junkfile, listing, mainfile, mainfile2, pairs, paths, picfile, record, results, results.old, subject, traversed

It is recommended that the user compile the program before submitting it to ASSET in order to insure that it is syntactically correct.

In addition, ASSET accepts certain Berkeley Pascal constructs which are not part of standard Pascal, including separate compilation (see section 5).

ASSET places certain arbitrary restrictions on the size of the program. Specifically,

number of variables visible to subject procedure <= 300
number of procedures and functions visible to subject procedure <= 300
number of types visible to subject procedure <= 300
number of constants visible to subject procedure <= 300
number of identifiers visible to subject procedure <= 500
length of each identifier <= 80
total number of characters in all identifiers visible to subject procedure <= 3000
number of basic blocks in subject procedure <= 100

For instructions on how to increase these limits, see Appendix I.

BUGS:

1. ASSET requires that the last statement in a case statement be followed by a semicolon. For example, the (syntactically correct) program fragment

```
case b of
      true: writeln('hello');
      false: writeln('goodbye')
```

```
                end
```
will be rejected by ASSET, but
```
                case b of
                        true: writeln('hello');
                        false: writeln('goodbye');
                end
```
will be accepted.

The user should check that all case statements in the subject procedure are of this form *before* submitting the program to ASSET.

2. ASSET translates **for** loops incorrectly: If a statement of the form

"**for** i := e1 **to** e2 **do** S"

appears in the subject procedure, in writing the modified subject procedure ASSET will translate this to a code fragment functionally equivalent to
```
       i := e1;
 <label1>:    if i > e2 then goto <label2>
              S;
              i := succ(i);
              goto <label1>;
<label2>:     next statement;
```
It should translate it to a fragment functionally equivalent to
```
        i := e1;
        tmp := e2;
<label1>:   if i > tmp then goto <label2>
            S;
            i := succ(i);
            goto <label1>;
<label2>:     next statement;
```
Thus, if any variable occurring in expression is modified within the statement S (which is legal, though poor programming practice), the modified subject program will not be equivalent to the subject program.

These bugs will be corrected in the next version of ASSET.

3. THE MOST IMPORTANT ASSET COMMANDS

In this section we describe the basic ASSET commands. The commands are listed in an order in which they can be executed. The section ends with some comments on the relationship between the commands. Additional commands are described in the next section.

Invoking ASSET

We will refer to the directory from which ASSET is invoked as the "current working directory". Before invoking ASSET you should set up a "subject directory" which contains the source of the program to be tested. It is usually convenient to have the subject directory be a subdirectory of current working directory, but this is not necessary. The subject directory *should not* be the same as the current working directory. To enter ASSET type "asset" in response to the UNIX prompt. ASSET will respond with

```
                Welcome to ASSET. For help type "help."
```

Enter relative pathname of initial default directory. >>:

At this point you should enter the pathname of the subject directory, relative to the current working directory. ASSET will then print the ASSET "command level prompt", ">>>: ", signifying that it is ready to accept a command.

We now describe each of the ASSET commands. These commands can be invoked when ASSET types the command level prompt. Some of these commands will query the user for additional information. When this occurs, ASSET will prompt the user with its "inner prompt", ">>: ". Sometimes ASSET will ask the user a question which has a yes or no answer. The user should respond with "Y" or "y" for yes and "N" or "n" for no. ASSET will indicate the default (if there is one) in brackets. If the default is "Y" then any non-null response which does not begin with "N" or "n" will be considered to be "Y". Similarly, if the default is "N" then any non-null response which does not begin with "Y" or "y" will be considered to be "N". If the user just hits carriage return, the inner prompt will continue to appear until a non-null response is given. The logical dependence of ASSET commands is given in Figure 1. The commands must be typed in lower case letters. Any command can be abbreviated by its first three letters.

begin

The "begin" command analyzes the subject procedure, breaking it into basic blocks, constructing its flow graph, and determining the blocks in which each variable has definitions and c-uses and the edges on which each variable has p-uses. An adjacency list for the graph is written to the file "graph", and a representation of the data-flow information is written to the file "defuse". The "String Table" in which ASSET has recorded the identifiers appearing in the subject program is written to the file "StringTabFile".

In addition, ASSET constructs a modified version of the subject program in which probes have been inserted. That is, at the beginning of basic block i, the statement "writeln(traversed,i :FW);" is inserted. Declarations of the text file "traversed", the constant "FW", and any labels which appear in the modified program are also inserted. We will refer to the resulting program as the "modified subject program". ASSET writes a pretty-printed version of the modified subject program to the file "copy.p".

ASSET also creates files "copy", "copy2" (preliminary versions of the modified subject program) and "SymTabFile" (a representation of the Symbol table for the subject program) which will not be used by subsequent commands, but may be useful for debugging.

When you type "begin", ASSET will respond with

Enter name of subject procedure file. >>:

If there is no file of this name in the subject directory ASSET will print an error message and return to the command level. Otherwise ASSET will copy the file to the file "subject", then say

Separate Compilation? (Y/N)[N] >>:

Answer "y" if the subject program is separately compiled, "n" otherwise. We will assume throughout the rest of this section that the subject program is NOT separately compiled. See section 5 for instructions on using ASSET on separately compiled programs.

Next, ASSET will ask you for the name of the procedure to be instrumented for testing. You may either supply the name of the procedure, or direct ASSET to prompt you with the names of the procedures in the program.

select

The "select" command causes ASSET to print a menu and query the user to select one of the data flow testing criteria.

find-associations

This command causes ASSET to produce a list of all definition-use associations (of the appropriate type) in the subject procedure. If the selected criterion is All-du-paths, find-associations writes a list of all of the du-paths in the subject procedure to the file "paths". Otherwise, find-associations writes a list of all of the definition-c-use associations and definition-p-use associations in the subject procedure to the file "pairs". Note that in the worst case, finding all of the du-paths may take time exponential in the number of basic blocks.

compile

The "compile" command invokes the Pascal compiler to compile the modified subject program. The executable code is written to the file "copy.exec". Warning and Error messages generated by the compiler are written to the file "comp.msgs".

run

The "run" command executes the modified subject program, on one or more test cases and adds the program trace to the file "traversed". When you type "run", ASSET may respond with

```
File 'traversed' already exists. Type 'Y' if you want to append to it, 'N'
otherwise. >>:
```

Type "Y" if you want to add the program traces generated by previous test cases to the current one, "N" otherwise. If you answer no, ASSET will ask you whether you want to save the old file "traversed" (i.e. the program traces of previous test cases).

Next, ASSET will ask you whether the program takes input from command line arguments. Answer yes if the subject program reads command line arguments via the Berkeley Pascal argv mechanism or if you would like to redirect the standard input or output using the UNIX redirection mechanism. If you answer yes, ASSET will prompt you for the command line arguments. For example supplying

```
<infile >outfile
```

as the command line argument will cause the modified subject program to read its standard input from the file "infile" and write its standard output to the file "outfile".

ASSET will then say

```
Executing modified subject program ...
```

then will execute the modified subject program with the command line arguments you've supplied. When execution terminates, ASSET will ask you whether you would like to run the program on additional test data. If you answer yes, ASSET will again ask you for command line arguments and execute the program. The new program trace will be added to the traces which have previously been written to the file "traversed". If you answer no, ASSET will return to the command level.

Note that one trace is produced for each run of the *subject program*, not one for each invocation of the *subject procedure*. Thus a single program trace may contain several paths from the entry to the exit of the subject procedure. If the subject procedure is recursive these paths will be "nested" within one another.

check

The "check" command checks whether the selected criterion is satisfied by the test data whose program traces are in the file "traversed". ASSET writes to the file "results" a list of those def-use associations which are required by the criterion but have not been covered by the test data. (Before doing this, ASSET moves the old version of the file "results" to the file "results.old". This allows the user to easily see which def-use associations were eliminated by the most recent additions to the test set by using the UNIX command "diff results results.old"). If the file "traversed" does not exist, ASSET initializes it to represent an empty set of program traces. In this case, "results" will contain a list of all of the def-use associations required by the criterion.

save

The "save" command moves the files "subject", "copy.p", "copy.exec", "traversed", "results", "pairs", "paths", "defuse", etc. from the current working directory to the subject directory.

exit

Moves important files to the subject directory and removes the rest, then exits.

Note on the relationship between the commands

Before checking whether a set of test data satisfies any of the criteria All-defs, All-p-uses, All-c-uses, All-c-uses/some-p-uses, All-p-uses/some-c-uses or All-uses it is necessary for ASSET to create a list of definition-use associations in the file "pairs". Before checking whether a set of test data satisfies the criterion All-du-paths it is necessary for ASSET to create a list of du-paths in the file "paths". When the most recently selected criterion is All-defs, All-p-uses, All-c-uses, All-c-uses/some-p-uses, All-p-uses/some-c-uses or All-uses the "find-associations" command creates the file "pairs". When the most recently selected criterion is All-du-paths the "find-associations" command creates the file "paths". Thus it is necessary to use the "find-associations" command *once* before the first time that "check" is invoked with one of the criteria involving "pairs" and *once* before the first time that "check" is invoked with the criterion du-paths.

Suppose the user has been attempting to satisfy criterion C then decides to switch to another criterion, C'. Let T be the set of test data on which the program has just been run (i.e. assume that set of program traces in the file "traversed" corresponds to test set T). Then to see how close T comes to satisfying C' the user need only invoke the "select-criterion" command (to select the new criterion, C') and the "check" command. It is not necessary to run the program on test set T again.

4. OTHER ASSET COMMANDS

csh

Spawns a new shell from which you can invoke your favorite UNIX commands. To exit from this shell and return to the ASSET command level type control-D.

directory

Displays the name of the subject directory and prompts the user for a new subject directory.

graphics

The "graphics" command writes a PIC program describing a graphical representation of the subject procedure's flow graph. The PIC program is written to the file "picfile". To obtain a picture of the flow graph, run the troff preprocessor "pic" (from a UNIX shell, not directly from ASSET) with picfile as input then pipe it through troff.

[At NYU, pipe the pic output through itroff (for output on a Canon Laser Printer), psroff (for output on an Apple Laser Writer), or preroff (for output on the console if you are in the Suntools environment.) For example, typing

```
% pic picfile | psroff -Pap3
```

(where "%" is the UNIX prompt) will send printed output to the LaserWriter in room 505.]

help

On line help facility.

purge

Removes (i.e. destroys) all of the files "subject", "copy.p", "copy.exec", "traversed", "results", "pairs", "paths", "defuse", etc. from the current working directory. Handle with care.

sepcomp

Compiles and links separately compiled subject program. See section 5.

restore

Copies the files "subject", "copy.p", "copy.exec", "traversed", "results", "pairs", "paths", "defuse", etc. from the subject directory to the current working directory, thus restoring a previous ASSET session. If the files in the subject directory were created by running ASSET on a separately compiled subject program some additional action is taken (see section 5.)

view

The "view" command directs ASSET to display the contents of a (text) file on the screen, a screenful at a time. If you type "view <filename>" ASSET will display that file. Otherwise, ASSET will prompt you for the file name. The view command works by invoking the UNIX command "more", hence responds to the same inputs as does "more".

Most importantly, type carriage return to display the next line of text, space to display the next screenful of text and q to return to the ASSET command level. For other options, see the UNIX manual page for "more".

5. USING ASSET ON SEPARATELY COMPILED PROGRAMS

This section describes how to use ASSET to test separately compiled programs. This section assumes some familiarity with Berkeley Pascal's separate compilation mechanism. (See the relevent sections of the Berkeley Pascal manual.)

To instrument a procedure for testing, ASSET must create modified versions of the file containing the source for that procedure, the file containing the source for the main program, and a file containing global variable declarations. To create an executable code for the modified subject program, ASSET must recompile and relink the modified files and any other files dependent on them.

In order to do this, ASSET assumes that subject directory contains a Makefile describing the dependency information (see documentation for the UNIX command "make"), that there is a file called "globals.h" which has global variable declarations, and such that the object files for the main program and for the procedure to be instrumented are dependent on globals.h.

If the user answers "yes" to the question "Separate compilation?" during execution of ASSET's "begin" command, ASSET prompts the user for the names of the file containing the main program and the file containing the subject procedure. We'll refer to these as <main_file> and <subject_proc_file>, respectively. ASSET moves <main_file>, <subject_proc_file>, and "globals.h" to <main_file>.bak, <subject_proc_file>.bak, and "globals.h.bak", respectively. ASSET writes the modified versions of the files to <main_file>, <subject_proc_file>, and "globals.h". Thus when the UNIX "make" command is invoked, the modified subject program will be compiled and linked.

To compile the modified subject program use the ASSET command "sepcomp", rather than the ASSET "compile" command. Sepcomp invokes the UNIX make command, and moves the executable code (which is assumed to be in "<subject-directory>/a.out") to copy.exec.

When the "save" command is invoked, ASSET looks for files with the ".bak" extension in the subject directory. For each such file, ASSET moves <fname> to <fname>.mod and moves <fname>.bak to <fname>. Thus the original files have the same contents as they had before the ASSET session, and the modified versions have the extension ".mod".

The "restore" command looks for files with the ".mod" extension and queries the user as to whether they should be restored. If the user answers yes to the question "restore <fname>.mod (Y/N)?" then ASSET moves <fname>to <fname>.bak and moves <fname>.mod to <fname>. Thus to restore a session in which the instrumented procedure was in file "Proc3.p", the main program was in file "main.p", answer yes to the questions "restore Proc3.p.mod?", "restore main.p.mod?", and "restore globals.h.mod?"

Restrictions on separately compiled programs :

The separate compilation option makes the following assumptions:

1. All relevant files are in the subject directory.
2. There is a file called "globals.h" containing global definitions in the subject directory.
3. "globals.h" is mentioned in #include directives in the subject procedure file and in the main program file.
4. There is a Makefile containing the dependency information, etc. needed to re-link and load the program contained in the subject directory.
5. The Makefile directs the executable code to file "a.out".
6. The procedure being instrumented is at the outer level of nesting.
7. The last included .h file cannot have any extra garbage at the end. (This is a bug).

NB -- It is currently NOT POSSIBLE to instrument the main program of a separately compiled module.

6. EXAMPLE OF AN ASSET SESSION

In this section we present an annotated example of an ASSET session. To distinguish between text written by the system and that written by the user, we display text entered by the user in boldface type. Comments are written in italics.

Example 1:

This example shows an ASSET session in which a brute-force string-matching procedure is analyzed. The program reads a string and a pattern. It is supposed to print the position in the string at which the pattern first appears, and print 0 if the pattern never appears in the string. The current working directory has a subdirectory called "StrMtch". The file "StrMtch/subject.p" contains the following program:

```
program TestStringMatch (input,output);
const MAX = 80;
        LENGTH = 10;
type  Source = array[1..MAX] of char;
        String = array[1..LENGTH] of char;
var   Pat : String;
      Txt : Source;
      i,result,TxtLen  : integer;

      function StringMatch(Pattern:String; SorText: Source;
            PatLen, SorLen : integer): integer;
      {Brute force pattern-matcher. Returns 0 for no match}
      var PatPos, SorPos : integer;
      begin
            PatPos := 1;
            SorPos := 1;
            repeat
                  if Pattern[PatPos] = SorText[SorPos] then
                  begin
                        SorPos := SorPos + 1;
```

```
                                        PatPos := PatPos + 1
                                end  {then}
                                else
                                begin
                                        SorPos := (SorPos - PatPos) + 2;
                                        PatPos := 1
                                end; {else}
                        until (PatPos > PatLen) or (SorPos > SorLen);
                        if PatPos > PatLen
                          then StringMatch := SorPos - PatLen
                          else StringMatch := 0
                end; {StringMatch}

        begin {main program}
                {Read Input}
                writeln('ENTER THE TEXT');
                i := 1;
                while (not(eoln) and (i <= MAX)) do
                begin
                        read(Txt[i]);
                        i := i + 1
                end;
                TxtLen := i - 1;
                readln;
                writeln('ENTER THE PATTERN');
                i := 1;
                while (not(eoln) and (i <= LENGTH)) do
                begin
                        read(Pat[i]);
                        i := i + 1
                end;
                readln;

                result := StringMatch(Pat,Txt,i - 1, TxtLen);
                writeln('The pattern first appears at position ',result:3,
                                ' in the text.');
        end.
```

```
Script started on Sun May 31 13:09:59 1987
csd27> asset

Welcome to ASSET. For help type "help."


Enter relative pathname of initial default directory.
>>: StrMtch

>>>: begin
```

```
Enter name of subject procedure file.
>>: subject.p
Separate Compilation? (Y/N) [N]
>>: n
Enter the name of the procedure to be instrumented.
If you would like to be prompted with the names of
the procedures in the subject program, just hit carriage return.
>>:

==> Should TestStringMatch
be instrumented for testing? (Y/N)
>>: n
==> Should StringMatch
be instrumented for testing? (Y/N)
>>: y

>>>: select

        SELECT A CRITERION

A. All-defs
B. All-c-uses
C. All-p-uses
D. All-c-uses/some-p-uses
E. All-p-uses/some-c-uses
F. All-uses
G. All-du-paths

Enter letter representing the selected criterion
>>: a
Criterion is All-defs.

>>>: find
```

We next check whether the criterion has been satisfied with no test data. This is not necessary, but by doing this, we get a list of all of the def-c-use and def-p-use associations in the program.

```
>>>: check
ALL-DEFS:

Still must exercise at least one of the following def-clear paths:
with respect to          from      to
Pattern                  1       ( 3, 5)
Pattern                  1       ( 3, 4)

AND
Still must exercise at least one of the following def-clear paths:
with respect to          from      to
```

```
SorText                1    ( 3, 5)
SorText                1    ( 3, 4)

AND
Still must exercise at least one of the following def-clear paths:
with respect to        from      to
PatLen                 1      8
PatLen                 1    ( 6, 3)
PatLen                 1    ( 6, 7)
PatLen                 1    ( 7, 9)
PatLen                 1    ( 7, 8)

AND
Still must exercise at least one of the following def-clear paths:
with respect to        from      to
SorLen                 1    ( 6, 3)
SorLen                 1    ( 6, 7)

AND
Still must exercise at least one of the following def-clear paths:
with respect to        from      to
PatPos                 2      4
PatPos                 2      5
PatPos                 2    ( 3, 5)
PatPos                 2    ( 3, 4)

AND
Still must exercise at least one of the following def-clear paths:
with respect to        from      to
SorPos                 2      4
SorPos                 2      5
SorPos                 2    ( 3, 5)
SorPos                 2    ( 3, 4)

AND
Still must exercise at least one of the following def-clear paths:
with respect to        from      to
PatPos                 4      4
PatPos                 4      5
PatPos                 4    ( 3, 5)
PatPos                 4    ( 3, 4)
PatPos                 4    ( 6, 3)
PatPos                 4    ( 6, 7)
PatPos                 4    ( 7, 9)
PatPos                 4    ( 7, 8)

AND
Still must exercise at least one of the following def-clear paths:
```

```
with respect to          from     to
SorPos                   4      4
SorPos                   4      5
SorPos                   4      8
SorPos                   4    ( 3,  5)
SorPos                   4    ( 3,  4)
SorPos                   4    ( 6,  3)
SorPos                   4    ( 6,  7)

AND
Still must exercise at least one of the following def-clear paths:
with respect to          from     to
PatPos                   5      4
PatPos                   5      5
PatPos                   5    ( 3,  5)
PatPos                   5    ( 3,  4)
PatPos                   5    ( 6,  3)
PatPos                   5    ( 6,  7)
PatPos                   5    ( 7,  9)
PatPos                   5    ( 7,  8)

AND
Still must exercise at least one of the following def-clear paths:
with respect to          from     to
SorPos                   5      4
SorPos                   5      5
SorPos                   5      8
SorPos                   5    ( 3,  5)
SorPos                   5    ( 3,  4)
SorPos                   5    ( 6,  3)
SorPos                   5    ( 6,  7)

To look at these again use the command 'view results'.
```

Next we will compile the program and start running it on some test data. As the initial test data set, we select one element in which the pattern appears in the string and one element in which the pattern does not appear in the string.

```
>>>: compile
Compilation begins ...
Done, and successful.

>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: n
Do you want to save old 'traversed'? (Y/N) [N]
>>: n
Command line arguments? (Y/N) [Y]
```

```
>>: n

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quick
The pattern first appears at position  5 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: y
Command line arguments? (Y/N) [Y]
>>: n

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quack
The pattern first appears at position  0 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n

>>>: check
ALL-DEFS:
CRITERION SATISFIED

To look at these again use the command 'view results'.
```

The test set satisfies the all-defs criterion. We next check whether the same test set satisfies a stronger criterion, all-uses.

```
>>>: select

        SELECT A CRITERION

A. All-defs
B. All-c-uses
C. All-p-uses
D. All-c-uses/some-p-uses
E. All-p-uses/some-c-uses
F. All-uses
G. All-du-paths

Enter letter representing the selected criterion
>>: f
```

Criterion is All-uses.

```
>>>: check
ALL-USES
Still need to exercise all of the following of def-clear paths:

with respect to          from    to
PatPos                   2      4
SorPos                   2      4
SorPos                   5      8
PatPos                   2     ( 3,  4)
SorPos                   2     ( 3,  4)
PatPos                   4     ( 7,  9)
PatPos                   5     ( 7,  8)

To look at these again use the command 'view results'.
```

To aid in the selection of test data which cover the remaining def-use associations, the user can draw the flow graph (see Figure 2) and use "copy.p" to aid in labeling each node with the corresponding code. Notice that for each i, block i begins with the statement "write(traversed,i:FW);"

```
>>>: view copy.p
program TestStringMatch(traversed, input, output);

var
   traversed: text;
const
   MAX = 80;
   LENGTH = 10;
type
   Source = array [1..MAX] of char;
   String = array [1..LENGTH] of char;
var
   Pat: String;
   Txt: Source;
   i, result, TxtLen: integer;

   function StringMatch(Pattern: String; SorText: Source; PatLen, SorLen: integer): integer;

   label
     10;
   const
     FW = 4;
   var
     PatPos, SorPos: integer;

   begin
   write(traversed, 1: FW);
   write(traversed, 2: FW);
   PatPos := 1;
   SorPos := 1;
  10:
   write(traversed, 3: FW);
   if Pattern[PatPos] = SorText[SorPos] then begin
      write(traversed, 4: FW);
      begin
         SorPos := SorPos + 1;
         PatPos := PatPos + 1
      end
   end else begin
      write(traversed, 5: FW);
      begin
         SorPos := SorPos - PatPos + 2;
         PatPos := 1
      end
   end;
   write(traversed, 6: FW);
   if not ((PatPos > PatLen) or (SorPos > SorLen)) then
      goto 10;
   write(traversed, 7: FW);
   if PatPos > PatLen then begin
      write(traversed, 8: FW);
      StringMatch := SorPos - PatLen
   end else begin
   write(traversed, 9: FW);
      StringMatch := 0
   end;
   write(traversed, 10: FW);
   write(traversed, 11: FW)
  end; { StringMatch }

begin
  rewrite(traversed);
  writeln('ENTER THE TEXT');
  i := 1;
  while not eoln and (i <= MAX) do begin

    read(Txt[i]);
    i := i + 1

  end;
  TxtLen := i - 1;
  readln;
  writeln('ENTER THE PATTERN');
  i := 1;
```

```
   while not eoln and (i <= LENGTH) do begin

     read(Pat[i]);
     i := i + 1

   end;
   readln;
   result := StringMatch(Pat, Txt, i - 1, TxtLen);
   writeln('The pattern first appears at position ', result: 3, ' in the text.')

end. { TestStringMatch }
```

Examining the annotated flow graph, we see that in order to execute a path from 2 to 4 which is definition clear with respect to PatPos, a test case in which the first character of the string matches the first character of the pattern is needed. We run the program on such a test case, adding its trace to those produced by the test cases run previously.

```
>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: y
Command line arguments? (Y/N) [Y]
>>: n

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
The
The pattern first appears at position   1 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n

>>>: check
Still need to exercise all of the following of def-clear paths:

with respect to           from      to
SorPos                     5      8
PatPos                     4      ( 7, 9)
PatPos                     5      ( 7, 8)

To look at these again use the command 'view results'.
```

Examining the annotated flow graph, we see that in order to execute a path from 5 to 8 which is definition clear with respect to SorPos, a test case in which the pattern is the *null string is needed. We run the program on such a test case, adding its trace to those produced by the test cases run previously.*

```
>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: y
Command line arguments? (Y/N) [Y]
>>: n

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN

The pattern first appears at position   2 in the text.
```

Examining the program's output, we see that the program has reported that the null string first appears in position 2 of the string. This is an error! The reader should note that this bug is guaranteed to be detected by any test set which is adequate according to the all-uses criterion. Having discovered a bug, we save the ASSET session and prepare to report the error.

```
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n

>>>: save
```

Note that to cover the one remaining association, (4,(7,9),PatPos), we would have to include a test case in which the first k characters of the pattern matched the last k characters of the text, for some k s.t. $0 < k <$ the length of the pattern.

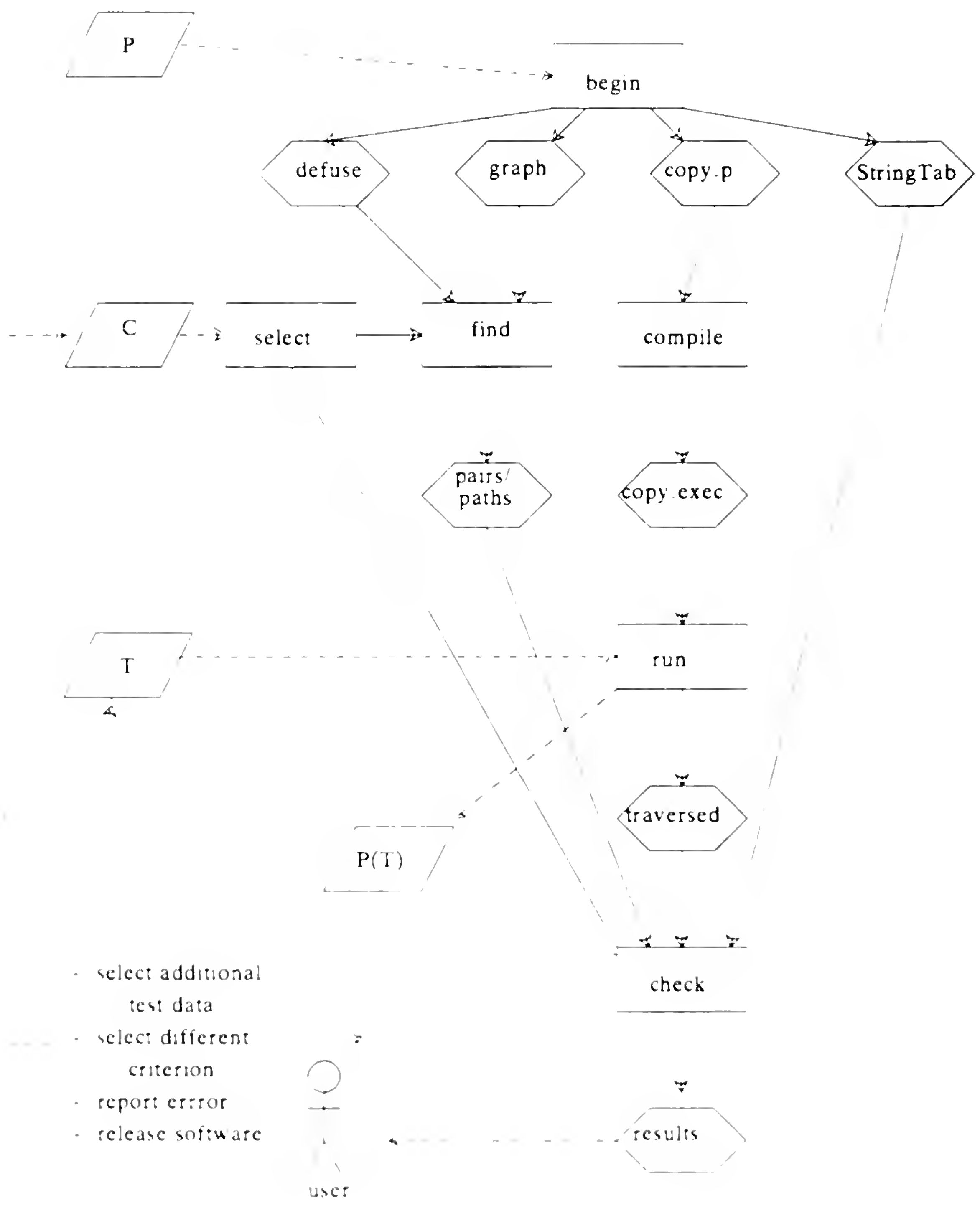
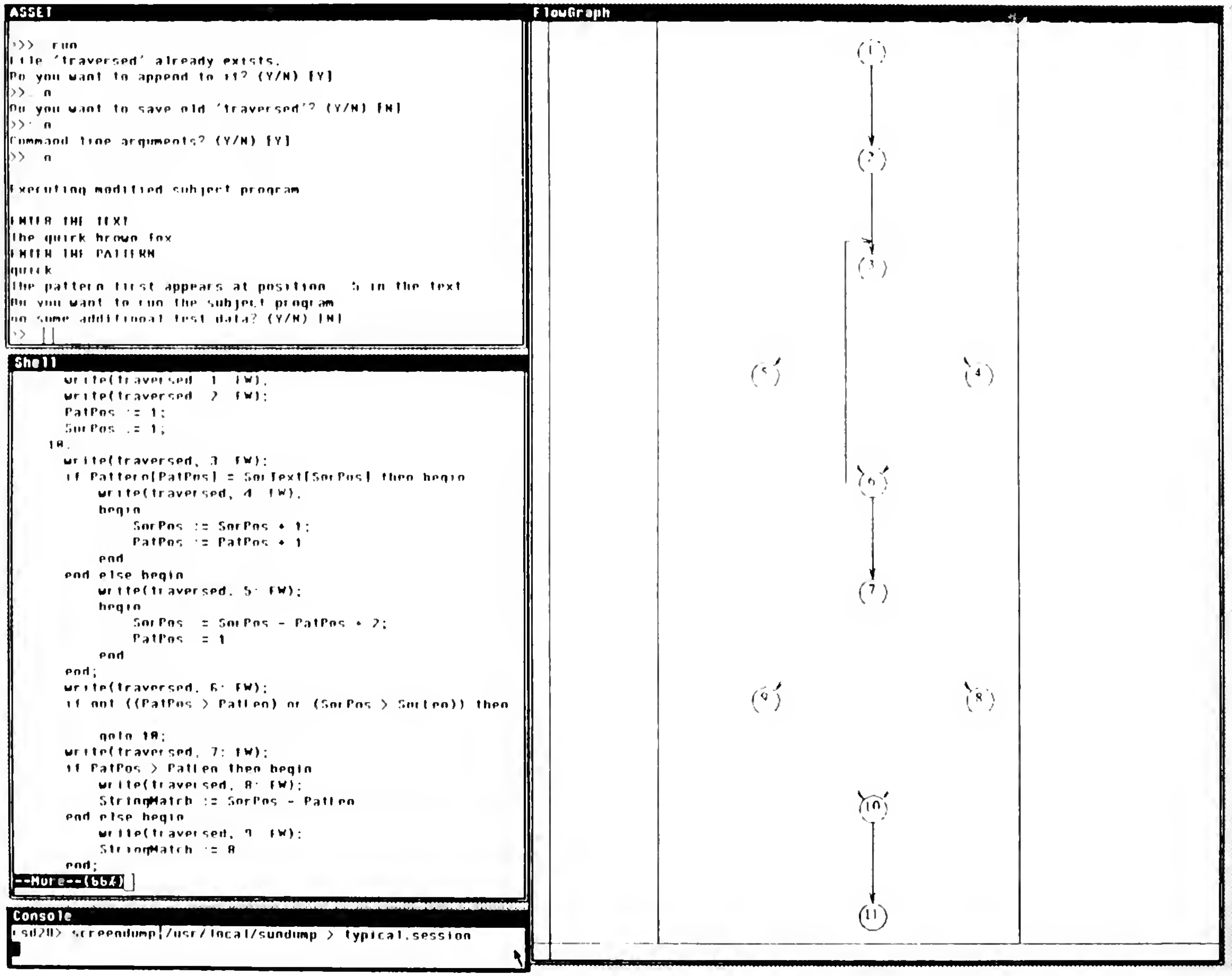


Figure 1
Schematic diagram of ASSET
P is program. C is criterion. T is Test Set.



```
>>  run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>  n
Do you want to save old 'traversed'? (Y/N) [N]
>>  n
Command line arguments? (Y/N) [Y]
>>  n

Executing modified subject program

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quick
The pattern first appears at position   5 in the text
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>
```

```
        write(traversed, 1, FW);
        write(traversed, 2, FW);
        PatPos := 1;
        SorPos := 1;
     10:
        write(traversed, 3, FW);
        if Pattern[PatPos] = SorText[SorPos] then begin
            write(traversed, 4, FW);
            begin
                SorPos := SorPos + 1;
                PatPos := PatPos + 1
            end
        end else begin
            write(traversed, 5, FW);
            begin
                SorPos := SorPos - PatPos + 2;
                PatPos := 1
            end
        end;
        write(traversed, 6, FW);
        if not ((PatPos > PatLen) or (SorPos > SorLen)) then

            goto 10;
        write(traversed, 7, FW);
        if PatPos > PatLen then begin
            write(traversed, 8, FW);
            StringMatch := SorPos - PatLen
        end else begin
            write(traversed, 9, FW);
            StringMatch := 0
        end;
--More--(66%)
```

```
sd20> screendump /usr/local/sundump > typical.session
```

7. REFERENCES

1. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE-11, No.4, April 1985, pp. 367-375.

2. P.G. Frankl, S.N. Weiss and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.

3. P.G. Frankl and E.J. Weyuker, "A Data Flow Testing Tool," *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.

4. P.G. Frankl and E.J. Weyuker, "Data Flow Testing in the Presence of Unexecutable Paths", *Proceedings of the IEEE Workshop on Software Testing*, Banff, Canada, July 1986.

APPENDIX II

Algorithm for derivation of initial path expression

In this appendix we present an algorithm which takes as input a flow graph G, and a definition-use association (d,u,v)* and which produces a path expression ξ such that all viable candidates are contained in PATHS(ξ). The length of the path expression ξ is O(N) where N is the number of nodes in the flow graph. The algorithm runs in time O(N). Throughout this section we assume that the flow graph G is formed according to the rules in Figure 2.1.1, from a subprogram having no **repeat-until** statements.

The algorithm is based on a generalization of the notion of dominators [HEC77]. Recall that node i *dominates* node j if and only if every path from the entry node to node j passes through i; i *properly dominates* j if and only if $i \neq j$ and i dominates j; i *immediately dominates* j iff

a) i properly dominates j, and

b) if k properly dominates j and $k \neq i$ then k (properly) dominates i.

We will say that node i *s-dominates* node j if and only if

a) i and j are reachable from s, and

b) every path from from s to j passes through i.

We will say that i *properly s-dominates* j if and only if $i \neq j$ and i s-dominates j; i *immediately s-dominates* j iff

a) i properly s-dominates j, and

* Actually, the algorithm makes no use of the variable v. However, it does assume that v has a definition in node d. As in chapter 5, we assume, without loss of generality, that u is a node, rather than an edge.

b) if k properly s-dominates j and k≠i then k (properly) s-dominates i.

We use s-DOM(j) to denote {i | i s-dominates j}.

Consider the graph G^s obtained from G by deleting all nodes and edges which are not reachable from s, deleting all edges of the form (i,s) and making s the entry node. It follows immediately from the definitions that i s-dominates j in G if and only if i dominates j in G^s. With this observation, the following properties of the s-dominance relation follow from the analogous properties of the dominance relation.

LEMMA AII.1: s-DOM(s) = {s}.

LEMMA AII.2: The s-dominance relation of a flow graph is a partial ordering.

LEMMA AII.3: Node s s-dominates all nodes which are reachable from s.

LEMMA AII.4: The s-dominators of a node form a chain (i.e. a linear ordering).

LEMMA AII.5: Every node except s which is reachable from s has a unique immediate dominator.

LEMMA AII.6: The graph of the reflexive and transitive reduction of the s-dominance relation of a flow graph is a forest. There is one tree T in this forest which contains all and only those nodes which are reachable from s. There is an arc (i,j) in T if and only if i immediately dominates j, and there is a path in T from i to j if and only if i dominates j.

We note that for a given s, all of the sets s-DOM(j) can be computed in time proportional to the number of nodes in G as follows.

1) Perform a depth first search on G starting at s to determine which nodes are in G^s and to derive a reverse postordering of the nodes in G^s.

2) Apply the dominator algorithm for reducible flow graphs [HEC77] to compute the sets DOM(j) in G^s. By the above observation, these are the sets s-DOM(j) in G.

Each of these steps takes O(e) bit vector operations, where e is the number of edges in G. Since G is a structured flow graph, e=O(|G|). To justify step 2, we note that if G^s were irreducible it would have a "forbidden subgraph" [HEC77]. Then G would also have a "forbidden subgraph". But since G is a structured flow graph, it is reducible, hence has no "forbidden subgraph."

Recall that given a definition-use association (d,u,v), algorithm PEM2 (Algorithm 5.5.2) needs to build up the path expression ξ, representing the initial set of candidates, from three components. $\alpha^{1,d}$ represents the set of paths from the entry node to node d; $\alpha^{d,u}$ represents the set of paths from node d to node u; $\alpha^{u,n}$ represents the set of paths from the entry node to node d;

We will present an algorithm which, given a "source" node s and a "target" node t, will produce a path expression which represents all of the paths $(n_1,...,n_k)$ where n_1=s, n_k=t, and n_j {s,t} for 1<j<k.* The intuition motivating the algorithm is that every path from s to t must pass through all of the s-dominators of t. In addition, such paths may pass through other nodes.

Recall that if node h is a loop header, then a *path through the loop headed by h* is a path of length greater than or equal to one, which begins and ends at node h, and which does not contain any nodes which are not in the loop headed by h. If n is not a loop header, we will say that a path π is *n-free* if and only if node n does not occur in π, except possibly as the first node in the path. If n is a loop header, we will say that π is n-free if and only if $\pi=\rho\sigma$, where ρ is either the null path or a path through the loop headed by n, and n does not occur in σ. For our purposes, given a source s and a target t, it will suffice

* The path expression produced may also represent some additional paths from s to t

to produce a path expression α such that all of the s-free paths from s to t belong to PATHS(α).

We will present two algorithms, BT1 and BT2. BT1 builds a path expression tree representing the s-free paths from s to t in the case that s dominates t. We first present the algorithm for BT1 and prove its correctness. We then show why this algorithm fails when s does not dominate t, and present a somewhat more complicate algorithm BT2, which handles the general case. Finally, we argue that BT2 is correct, analyze its complexity, and show that the length of the path expression produced is O(N).

We distinguish between three types of nodes which may occur in the flow graph G: loop header nodes, selection exit nodes, and regular nodes. A *loop header* node, n, has exactly two predecessors, p_1 and p_2, where p_1 is the immediate dominator of n, and n dominates p_2. A *selection* exit, has at least two predecessors; if n is a selection exit then none of n's predecessors dominate n. A *regular* node has at most one predecessor; if p is the predecessor of a regular node n, then p is the immediate dominator of n. It follows from the structure of the subgraphs given in Figure 2.1.1 that the sets {loop header nodes}, {selection exit nodes}, and {regular nodes} form a partition of the set of nodes. The nodes in G can easily be classified by performing a depth first search of G. For each back edge, (i,j), j is a loop header; for each cross edge, (i,j), j is a selection exit; all nodes which are neither the target of a back edge, nor the target of a cross edge are regular nodes.

The algorithms BT1 and BT2 make use of the following three primitive functions. CONCAT takes as input two trees, T_1 and T_2, and returns the tree whose root is labelled by the symbol "$\cdot$", whose left subtree is T_1, and whose right subtree is T_2. UNION takes as input two trees, T_1 and T_2, and returns the tree whose root is labelled by the symbol

"$\bigcup$", whose left subtree is T_1, and whose right subtree is T_2. MAKEVERTEX takes as input a node symbol $\mathbf{n}_i$ or a loop symbol $\mathbf{p}_i$, and returns the tree whose root is labelled by that symbol, and whose left and right subtrees are null.

Function BT1 is given in Figure AII.1.

```
function BT1(s,t: NodeRange): Tree;
{returns path expression tree representing set of paths from s to t, where s dominates t}
begin
      if s does not dominate t then Error;
      if s=t then BT1 := MAKEVERTEX(λ)
      else
      begin
            imm_s_dom := immediate s-dominator of t;
            if t is regular node then
                  BT1 := CONCAT(BT1(s, imm_s_dom),MAKEVERTEX(n_t))
            else if t is a loop header then
                  BT1 := CONCAT(BT1(s, imm_s_dom),MAKEVERTEX(p_t))
            else {t is a selection exit}
            begin
                  let i_1,...,i_m be the predecessors of t;
                  {m is greater than or equal to 2}
                  T1:= UNION(BT1(imm_s_dom,i_1),
                        BT1(imm_s_dom,i_2));
                  for j := 3 to m do
                  begin
                        T2 := T1;
                        T1 := UNION(BT1(imm_s_dom,i_j),T2);
                  end;
{*}               T2 := CONCAT(T1,MAKEVERTEX(n_t));
                  BT1 := CONCAT(BT1(s, imm_s_dom),T2);
            end
      end
end.
```

Figure AII.1

Roughly speaking, BT1 involves climbing upwards through the tree of immediate-s-dominators from t to s. Whenever a selection-exit n is encountered, BT1 calls itself recursively to build subtrees representing the sets of paths from the entrance of the selection subgraph to the predecessors of n, then "unions" these subtrees together.

THEOREM

Let s dominate t. Let α be the path expression represented by the tree returned by BT1(s,t). Then if node s is not a loop header, the path expression $\mathbf{n}_s \cdot \alpha$ represents the set of s-free paths from s to t, and if node s is a loop header, the path expression $\mathbf{p}_s \cdot \alpha$ represents the set of s-free paths from s to t.

PROOF:

Let $G^{s,t}$ be the subgraph of G^s consisting of those nodes from which t is reachable. The proof is by course-of-values induction on the size of the $G^{s,t}$.

Base Case: $|G^{s,t}|=1$.

If $|G^{s,t}|=1$, then s=t, so BT1(s,t) consists of a single vertex labelled by $\boldsymbol{\lambda}$, and $\alpha=\boldsymbol{\lambda}$. PATHS($\mathbf{n}_s \cdot \alpha$)=PATHS($\mathbf{n}_s$)=$\{(n_s)\}$, which is the set of paths from s to t.

Inductive case:

Suppose the theorem holds for all s' and t' such that s' dominates t' and $|G^{s',t'}|< k$. We show that the theorem holds for s and t such that $G^{s,t}= k$. We will assume that s is not a loop header. The proof when s is a loop header is identical except that occurrences of the symbol $\mathbf{n}_s$ are replaced by the symbol $\mathbf{p}_s$.

case 1: t is a regular node.

Let d be the immediate s-dominator of t. Every s-free path from s to t is of the form πn_t where π is an s-free path from s to d. The function BTW(s,t) returns a tree representing the path expression $\alpha=\beta \cdot \mathbf{n}_t$, where β is the path expression represented by BT1(s,d). So $\mathbf{n}_s \cdot \alpha = \mathbf{n}_s \cdot \beta \cdot \mathbf{n}_t$. By the inductive hypothesis, $\mathbf{n}_s \cdot \beta$ represents the set of s-free paths from s to d. Therefore, PATHS($\mathbf{n}_s \cdot \alpha$)={$\pi n_t$| π is an s-free path from s to d}={s-free paths from s to t}.

case 2: t is a loop header.

Let d be the immediate s-dominator of t. Every s-free path from s to t is of the form $\pi\rho$ where π is an s-free path from s to d and ρ is a path through the loop headed by node t. The function BTW(s,t) returns a tree representing the path expression $\alpha=\beta \cdot \mathbf{p}_t$, where β=BT1(s,d). So $\mathbf{n}_s \cdot \alpha = \mathbf{n}_s \cdot \beta \cdot \mathbf{p}_t$. By the inductive hypothesis, $\mathbf{n}_s \cdot \beta$ represents the set of s-free paths from s to d. By definition, $\mathbf{p}_t$ represents the set of through the loop headed by t. Therefore, PATHS($\mathbf{n}_s \cdot \alpha$)={s-free paths from s to t}.

case 3: t is a selection exit.

Let d be the immediate dominator of t. Note that d is the entrance to the selection subgraph corresponding to t. Since s dominates t, s also dominates d, so d is the immediate s-dominator of t. Let $i_1,...i_m$ be the predecessors of t. Note that d dominates each of the i_j. Every s-free path from s to t is of the form $\rho d \sigma t$, where ρd is a s-free path from s to d, and $d\sigma$ is a d-free path from d to one of the i_j.

By the inductive hypothesis, for each j, $1 \le j \le m$, the path expression α_j represented by the tree BT1(d,i_j), has the property that $\mathbf{n}_d \cdot \alpha_j$ represents the set of d-free paths from d to i_j. So after executing the statement marked by [*], T2 represents a path expression β, where $\mathbf{n}_d \cdot \beta$ represents the set of d-free paths from d to t. By the inductive hypothesis, BT1(s,d) represents a path expression γ where $\mathbf{n}_s \cdot \gamma$ represents the set of s-free paths from s to d. So BT1(s,t) represents the path expression $\gamma \cdot \beta$ where $\mathbf{n}_s \cdot \gamma \cdot \beta$ represents the set of s-free paths from s to t.■

When s does not dominate t, the situation becomes more complicated. Intuitively, this is because the graph G^s is not necessarily a structured flow graph. In this situation the immediate s-dominators of a node is not necessarily the same as its immediate dominator.

Consider the flow graph shown in Figure AII.2.a. Let s be node 6. The tree of immediate 6-dominators is shown in Figure AII.2.b. Those nodes which are loop headers (in G) have been marked with an asterisk, and those which are selection exits (in G) are marked with a plus sign.

Let t be node 10. Climbing upward through the tree of immediate-6-dominators from 10 to 6 yields the path expression $\alpha=\mathbf{6}\cdot\mathbf{8}\cdot p_4\cdot\mathbf{9}\cdot\mathbf{10}$. This path expression represents some, but not all, of the paths from node 6 to node 10. It does not represent those paths which, after visiting node 9, "bypass" node 10, going instead to nodes 11 and 12, then around the loop headed by node 2 zero or more times, then back to nodes 3, 4, and 9. We can view the problem as arising from the fact that node 4 can be preceded either by node 8, which immediately-s-dominates node 4, or by node 3, which does not s-dominate node 4. Note that the immediate 6-dominator of node 4 and the immediate dominator of node 4 differ. If we replace the symbol p_4 in α by the sub-path-expression

$$(p_4\cdot\mathbf{9}\cdot(\mathbf{10}\cup\mathbf{11})\cdot\mathbf{12}\cdot p_2\cdot\mathbf{3}\cup\lambda)\cdot p_4,$$

which represents the set of paths from node 4 to node 4, we obtain the expression

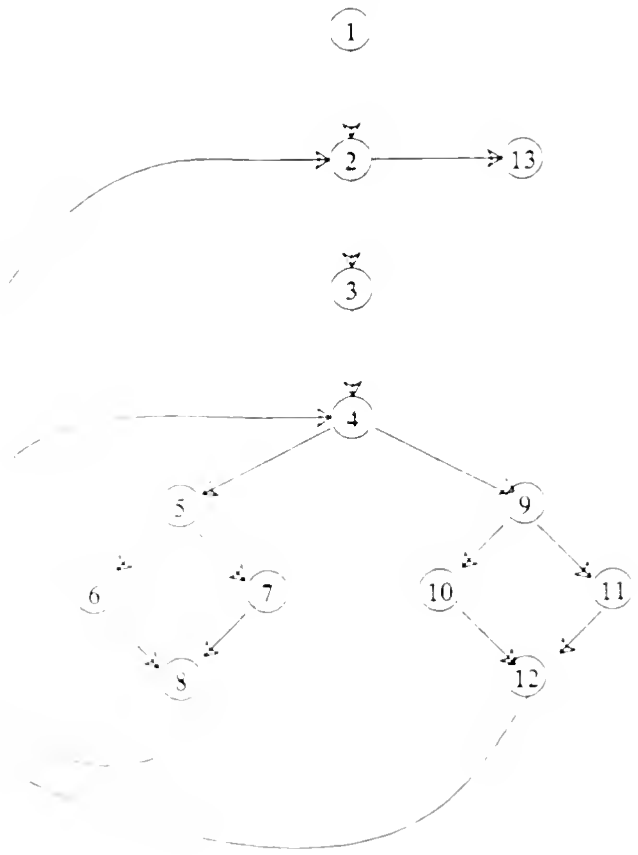
$$\mathbf{6}\cdot\mathbf{8}\cdot\ (p_4\cdot\mathbf{9}\cdot(\mathbf{10}\cup\mathbf{11})\cdot\mathbf{12}\cdot p_2\cdot\mathbf{3}\cup\lambda)\cdot p_4\ \cdot\mathbf{9}\cdot\mathbf{10},$$

which represents the set of paths from node 6 to node 10, as desired.

A similar problem arises in the flow graph shown in Figure AII.3.a. Let s be node 4 and let t be node 8. The tree of immediate 4-dominators is shown in Figure AII.3.b. Climbing up the tree from node 8 to node 4 gives the path expression $\mathbf{4}\cdot\mathbf{6}\cdot\mathbf{7}\cdot\mathbf{8}$, which represents some, but not all of the paths from 4 to 8. If we replace the symbol **6** by the sub-path-expression

FIGURE AII.2

(a)



(b)

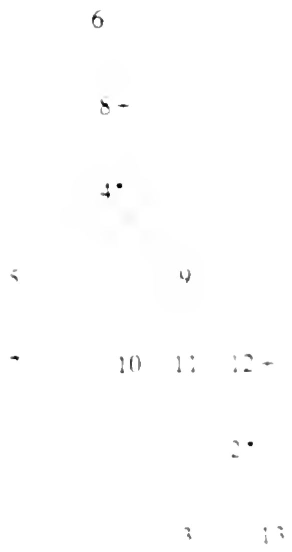
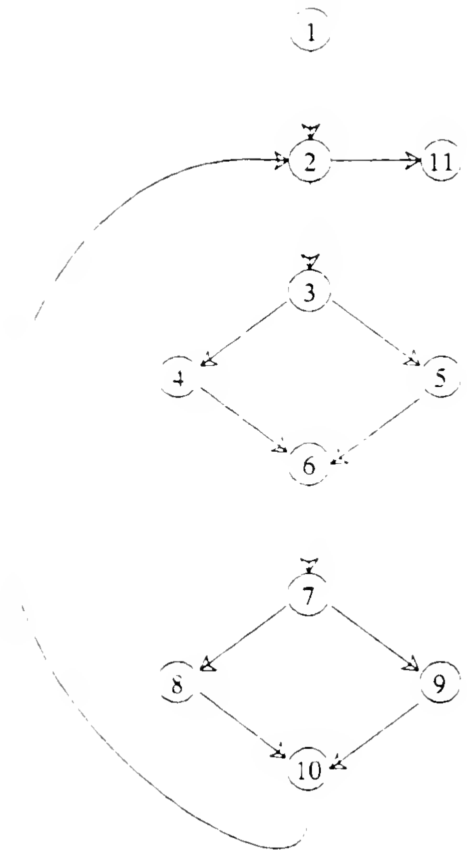
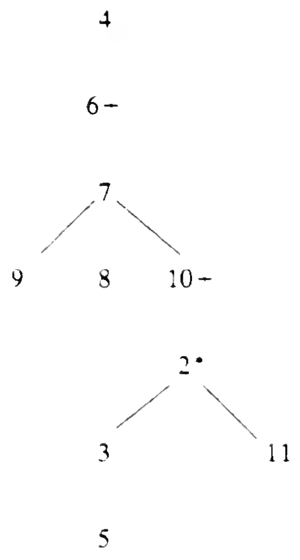


FIGURE AII.3

(a)



(b)



$$\mathbf{6 \cdot (7 \cdot (8 \cup 9) \cdot 10 \cdot p_2 \cdot 3 \cdot (4 \cup 5) \cdot 6 \cup \lambda)},$$

which represents the set of paths starting and ending at node 6, we obtain the path expression $\mathbf{4 \cdot 6 \cdot (7 \cdot (8 \cup 9) \cdot 10 \cdot p_2 \cdot 3 \cdot (4 \cup 5) \cdot 6 \cup \lambda) \cdot 7 \cdot 8}$, which represents the set of paths from node 4 to node 8, as desired.

We capture these idea algorithmically, as follows. As before, BT2 involves climbing upwards through the tree of immediate s-dominators from t to s. If the node being processed is a regular node, a loop header whose immediate s-dominator is the same as its immediate dominator, or a selection exit whose immediate s-dominator is the same as its immediate dominator, then BT2 will behave like BT1. However if the node n being processed is a loop header or selection exit whose immediate s-dominator and immediate dominator differ, it may be necessary to take special action, in order to include a sub-path-expression representing the set of n-free paths from n to n. As we shall see below, this special action is not always necessary, because the sub-path-expression produced may be redundant. We will present the algorithm for function BT2, after some preliminary definitions.

Let s be a node and let n be a loop header node in G. Let e be the exit node of the loop headed by n and let d be the immediate dominator of n. If d is reachable from s and e s-dominates n, we will call n an *s-inverted loop header*. Note that if n is s-inverted then s is in the loop headed by n and s≠n. If n is s-inverted then the immediate dominator of n and the immediate s-dominator of n are different nodes. Thus, the loop in G^s headed by n is different than the loop in G headed by n.

LEMMA AII.7:

Let n be an s-inverted loop header and let d be its immediate dominator. Then there is exactly one loop header h such that

1) n is in the body of the loop headed by h, and

2) n properly s-dominates h and h s-dominates d.

Note that h may actually be d itself. We will call h the *header corresponding node n.*

PROOF:

Since n is s-inverted, s is in the body of the loop headed by n. Otherwise, every path from s to e would pass through n. Since d is reachable from s there must be some loop which contains both d and s (see Figure AII.4.) Let h be the header of the innermost such loop. It is immediately evident from the figure that h satisfies 1) and 2).

Let $h' \neq$ h be another header which satisfies 1). Then the loop headed by h is nested within the loop headed by h'. There is a path from s to d which does not pass through h', so h' does not satisfy 2).■

We will now present the second algorithm, BT2, and argue that it is correct.

FIGURE AII.4

s

e

n•

h•

d

```
procedure BT2(s1,t: NodeRange; var HeaderHit: boolean; var T: Tree);
var imm_dom, imm_s_dom :NodeRange;
begin
{1}  if s1 dominates t then T := BT1(s1,t)
{2}  else begin
{3}      imm_dom := immediate dominator of t;
{4}      imm_s_dom := immediate s-dominator of t;
{5}      if t is a regular node then
         begin
{6}          BT2(s1,imm_s_dom,HeaderHit,T1);
{7}          T := CONCAT(T1,MAKEVERTEX($n_t$))
         end
{8}      else if t is a loop header then
         begin
{9}          mark that t has been visited;
{10}         if (t is inverted loop header) then
             begin
{11}             HeaderHit := true;
{12}             if the header associated with t has not yet been visited then
                 begin
{13}                 BT2(t,imm_dom,HeaderHit,T1);
{14}                 T2  := UNION
                            (CONCAT(MAKEVERTEX($p_t$),T1)
                            MAKEVERTEX($\lambda$));
{15}                 T3  := CONCAT(T2,MAKEVERTEX($p_t$));
{16}                 BT2(s1,imm_s_dom,HeaderHit,T4);
{17}                 T := CONCAT(T4,T3);
                 end
{18}             else
                 begin
{19}                 BT2(s1,imm_s_dom,HeaderHit,T1);
{20}                 T := CONCAT(T1,MAKEVERTEX($p_t$));
                 end
             end
{21}         else
             begin
{22}             BT2(s1,imm_s_dom,HeaderHit,T1);
{23}             T := CONCAT(T1,MAKEVERTEX($p_t$));
             end
         end
{24}     else { t is a selection exit }
{25}         if imm_dom s-dominates t then
{26}             BT2(s1,imm_s_dom,HeaderHit,T1);
{27}             T := CONCAT(T1,BT1(imm_dom,t))
{28}         else if t s-dominates imm_dom and not HeaderHit then
             begin
{29}             T1 := BT1(imm_dom,t);
{30}             BT2(t,imm_dom,HeaderHit,T2);
```

```
{31}                    T3  := CONCAT(T2,T1);
{32}                    T4  := UNION(MAKEVERTEX(λ),T3);
{33}                    T5  := CONCAT(MAKEVERTEX(n_t),T4);
{34}                    BT2(s1,imm_s_dom,HeaderHit,T6);
{35}                    T := CONCAT(T6,T5)
                    end
{36}                else
                    begin
{37}                    BT2(s1,imm_s_dom,HeaderHit,T1);
{38}                    T := CONCAT(T1,MAKEVERTEX(n_t))
                    end
        end
end.
```

```
function BuildTree(G,s,t : NodeRange) : Tree;

begin
        {initialize global variables}
        build tree of immediate dominators;
        build tree of immediate s-dominators;
        classify each node as being a regular node, a loop header, or a selection exit;
        mark each s-inverted loop header as such and find the associated loop header;
        mark each loop header as  unvisited;
        HeaderHit := false;

        {call procedure BT2 to build tree T=$T^\alpha$,
         where $n_s \cdot \alpha$ represents the set of candidates corresponding
         to the definition-use pair (s,t)}
        BT2(s,t,HeaderHit,T);
        BuildTree := CONCAT(MAKEVERTEX(n_s),T);
end;
```

We now argue, somewhat informally, that BuildTree(G,s,t) returns a tree representing a set of paths which contains all of the s-free paths from s to t, as desired.

When BT2 is called from BuildTree, the effect is to "climb" upward (toward the root) through the tree of immediate s-dominators from t to s, backing down the tree under certain special circumstances.

We first discuss the cases where the immediate dominator and immediate s-dominator of the current node are the same. If the current node t is a regular node, then the algorithm "emits" a vertex labelled by $\mathbf{n}_t$ (line 7) and continues to climb up through the tree (line 6). Similarly, if the t is a non-inverted loop header node, then the algorithm "emits" a vertex labelled by $\mathbf{p}_t$ (line 23) and continues to climb up through the tree (line 22). If the t is a selection exit whose immediate dominator, d, and immediate-s-dominator are the same node, then the algorithm "emits" a subtree representing the d-free paths from d to t (line 27) then continues to climb up through the tree (line 26). In each of these cases, an argument similar to the ones made in the proof of correctness of BT1 could be made to show that the resulting path expression includes all of the s-free paths from s1 to t.

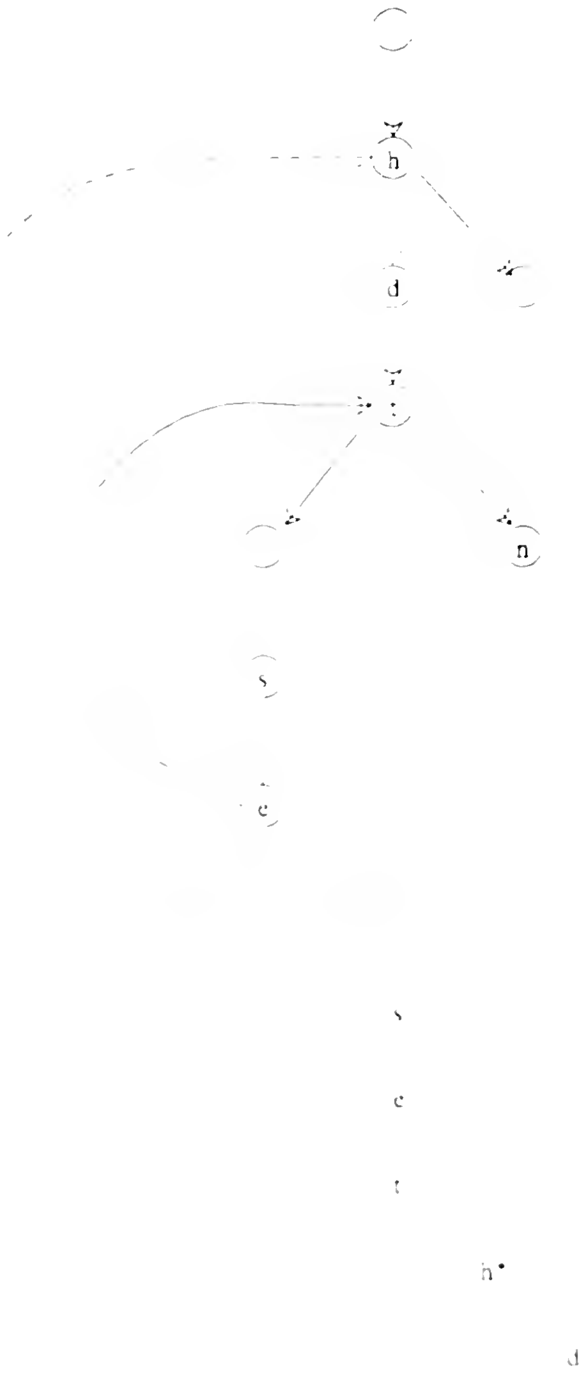
We next consider what happens when the current node, t, is an inverted loop header. Let d be the immediate dominator of t, let e be the other predecessor of t (the exit node of the body of the loop headed by t, hence the immediate-s-dominator of t) let h be the header associated with t, and let n be the successor of t which is not in the loop (see Figure AII.5.)

If node h, the header corresponding to t, has not yet been visited, then the algorithm returns a tree corresponding to the path expression $\gamma=\beta\cdot(\mathbf{p}_t\cdot\alpha\cup\lambda)\cdot\mathbf{p}_t$ where β comes from the recursive call to BT2 in line 16 and α comes from the recursive call to BT2 in line 13.

The s-free paths from s to t are all of the form $\pi\rho$, where π is an s-free path from s to e and ρ is either of the form $e\sigma\tau\zeta$ or $e\sigma$, where σ and ζ are s-free paths through the loop headed by t and τ is an s-free path from n to d. Any s-free path from s to t can be decomposed in such a way that $\pi\in$ PATHS(β) and $\rho\in$ PATHS($(\mathbf{p}_t\cdot\alpha\cup\lambda)\cdot\mathbf{p}_t$).

If node h, the header corresponding to t, has already been visited, then a subtree labelled by a loop symbol representing the outermost loop header which is reachable

FIGURE AII.5



from s and which reaches the original target has already been emitted. All of the paths from t to d are subpaths of paths represented by this loop header. So it would be redundant to emit the path expression $(p_t \cdot \alpha \cup \lambda) \cdot p_t)$. Thus, in lines 22 and 23 the algorithm emits a path expression representing the s-free paths which go from s to e, then through the loop headed by t.

If the current node t is a selection exit, there are four cases to consider. Let id represent the immediate dominator of t and let isd represent the immediate s-dominator of t. If id=isd, then s dominates id, and the situation is similar the situation treated in algorithm BT1. BT2 emits a path expression $\alpha \cdot \beta$, where α (the path expression corresponding to the tree returned by the recursive call in line 22) represents s-free paths from s to id and β (the path expression corresponding to the tree returned by the call to BT1 in line 23) represents paths through the selection subgraph.

If id=isd then node s is in the selection subgraph whose exit node is t. This subgraph has a different "shape" in G^s than in G. If t s-dominates id {line 28}, then there is a path from s to id. Each s-free path from s to t is of the form $\pi\rho$, where π is n s-free path from s to t and ρ is a path from t to t. Any path from t to t is either the null path or a path $\sigma\zeta$ where σ goes from t to id and ζ goes from id to t. The path expression corresponding to the tree built in lines 29 to 35 represents these paths. Intuitively, in lines 29 and 30 the algorithm "backs down" the tree of immediate s-dominators.

If the variable *HeaderHit* has already been set to true when control reaches line 28, then the outermost loop header containing s and the original target node has already been emitted, so the path expression representing paths from t to t is redundant. this case the algorithm emits a path expression (in lines 37 and 38) representing the s-free paths which go "directly" from s to t, without "backing down" the tree of immediate s-dominators.

Similarly, if id does not s-dominate t and t does not s-dominate id, then id is not reachable from s, so the algorithm emits a path expression (in lines 37 and 38) representing the s-free paths which go "directly" from s to t, without "backing down" the tree of immediate s-dominators.

In the execution of BuildTree(G,s,t) no vertex in the tree of immediate s-dominators is visited more that twice. This is because the only way a vertex which has already been visited can be visited again is if the algorithm "backs down" the tree of immediate-s-dominators in line 13 or line 30. When the recursive call in line 30 is performed, an inverted loop header node which is s-dominated by t and which s-dominates id is guaranteed to be visited, at which point *HeaderHit* will be set to true. So line 30 is executed at most once in the execution of BuildTree. When line 13 is executed, the headers corresponding to inverted loop header t and to *all of the other* inverted loop headers in whose loop t lies are visited. Thus in the execution of BuildTree, each edge (id,t) is backed down at most once. It follows that the length of the path expression produced by BuildTree is O(N) and that the running time of BuildTree is O(N) where N is the number of nodes in G.

APPENDIX III:

In this appendix we present a script of a session with an implemented prototype of the path expression heuristic. This implementation is interactive. It allows the user to select the loop symbol on which to concentrate. It also requires that the user perform the partial symbolic evaluation and theorem proving by hand. The implemented algorithm differs in certain minor ways from the algorithm described in chapter 5.

In this session we examine the association (2,(6,14),switch) from the bubblesort program in Figure 5.2.2. This is the same association which was examined in Examples 5.2.2 and 5.5.1.

Text entered by the user is printed in bold face. Comments are in italics. The symbol F stands for Ø and the symbol L stands for λ.

```
Script started on Tue Sep 22 21:10:05 1987
csd27> heuristic
Would you like to check another def-use association?(Y/N)
==> y
Enter the source and target node
2 14
Enter the variable
switch
The initial path expression is
N2.P3.N5.P6.N14

Select one of the following actions:
A: Expand a loop symbol
```

B: Toggle parentheses

C: Print current path expression

D: Print Always/Never info for subexpression

E: Give up

===>a

Enter position of a loop symbol

3

User directs system to first concentrate on the symbol P3, which is the third symbol in the current path expression.

Expanding loop symbol P3.

Use partial symbolic evaluation to derive the partial path condition E1& ... &Ek for the path expression

botP3.N5.P6.N14top

which follows the loop symbol.

Derive a predicate which is always true when control reaches the top of this occurrence of P3 by partially symbolically evaluating the path expression

botN1.N2.P3top

For each i,

If (Ei & C) is unsatisfiable, let MRDi be the set of variables occurring in Ei.

Otherwise MRDi is empty.

```
Enter the non-empty sets MRDi

Enter variables in the next MRDi set, one per line.
Type "." to end the list of variables in this set.
Enter "*" to end the list of sets.
i
.
Enter variables in the next MRDi set, one per line.
Type "." to end the list of variables in this set.
Enter "*" to end the list of sets.
*
```

User enters the set {i} because the predicate i>5 is necessary on exit from the loop and is unsatisfiable before entry into the loop. Thus i must change value on any executable path through the loop.

```
Expanding loop symbol

The simplified subexpression representing the loop is
((N3.N4)+ U L).N3

The new path expression is
N2.((N3.N4)+ U L).N3.N5.P6.N14

Select one of the following actions:
A: Expand a loop symbol
B: Toggle parentheses
```

C: Print current path expression

D: Print Always/Never info for subexpression

E: Give up

===>a

Enter position of a loop symbol

14

The loop symbol P6 is the 14th symbol in the path expression

(not counting parentheses.)

Expanding loop symbol P6.

Use partial symbolic evaluation to derive the partial

path condition E1& ... &Ek for the path expression

    botP6.N14top

which follows the loop symbol.

Derive a predicate which is always true when control

reaches the top of this occurrence of P6

by partially symbolically evaluating the path expression

    botN1.N2.((N3.N4)+ U L).N3.N5.P6top

For each i,

If (Ei & C) is unsatisfiable, let MRDi be the

set of variables occurring in Ei.

Otherwise MRDi is empty.

Enter the non-empty sets MRDi

```
Enter variables in the next MRDi set, one per line.
Type "." to end the list of variables in this set.
Enter "*" to end the list of sets.
n
switch
.
Enter variables in the next MRDi set, one per line.
Type "." to end the list of variables in this set.
Enter "*" to end the list of sets.
*
n or switch must be redefined on any executable path
through the loop
Expanding loop symbol
Pruning the expression. Variable
switch is always defined in the sub-expression
(N6.N7.P8.N13)+

The simplified subexpression representing the loop is
N6
Each path through the body of the loop has a definition of switch

None of the variables in the set
{ switch n }
are defined in the subexpression
N6
```

```
The new path expression is

F


No more candidates for unexecutable def-clear path.


The association is unexecutable


Would you like to check another def-use association?(Y/N)

==> n

csd27> ^D

script done on Tue Sep 22 21:15:41 1987
```

NYU COMPSCI TR-394 c.1
Frankl, Phyllis
The use of data flow
information for the
selection and...

DATE DUE	BORROWER'S NAME